FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

POPH QUALIFICAR É CRESCER.

QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL

GOVERNO DA REPÚBLICA PORTUGUESA

UNIÃO EUROPEIA
Fundo Social Europeu

Filipe Alexandre Andrade Salgado

FAT-DBT Engine (Framework for Application-Tailorcd, Co-designcd Dynamic Binary Translation Enginc)

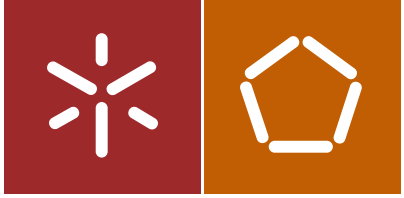
UMinho | 2017

Universidade do Minho
Escola de Engenharia

Filipe Alexandre Andrade Salgado

FAT-DBT Engine (Framework for Application-Tailorcd, Co-designcd Dynamic Binary Translation Enginc)

setembro de 2017



Universidade do Minho
Escola de Engenharia

Filipe Alexandre Andrade Salgado

FAT-DBT Engine (Framework for Application-Tailorcd, Co-designcd Dynamic Binary Translation Enginc)

Tese de Doutoramento
Programa Doutoral em
Engenharia Eletrónica e de Computadores (PDEEC)

Trabalho efetuado sob a orientação do
Professor Doutor Adriano José da Conceição Tavares
Professor Doutor José Araújo Mendes

setembro de 2017

DECLARAÇÃO

Nome: Filipe Alexandre Andrade Salgado

Correio electrónico: filipe.salgado@gmail.com

Tel./Tlm.: 934984144

Número do Bilhete de Identidade:13206863

Título da dissertação: **FAT-DBT Engine (Framework for Application-Tailored, Co-designed Dynamic Binary Translation Engine)**

Ano de conclusão: 2017

Orientadores:

Doutor Adriano José da Conceição Tavares

Doutor José Araújo Mendes

Designação do Doutoramento: Programa Doutoral em Engenharia Eletrónica e de Computadores (PDEEC)

Área de Especialização: Informática Industrial e Sistemas Embebidos

Escola: Escola de Engenharia

Departamento: Departamento de Electrónica Industrial

Assinatura: 

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.
I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 22/9/2017

Full name: Filipe Alexandre Andrade Salgado

Signature: Filipe Alexandre Andrade Salgado

Acknowledgments

Thank you very much to everyone who contributed to the completion of this thesis. I must mention my advisor Dr. Adriano Tavares for the valuable guidance through all this years, also Dr. Jorge Cabral and Professor João Monteiro for their support and Dr. Mongkol Ekpanyapong for his advise during my stay abroad. More than owing great respect to all of you, allow me to consider you great friends of mine also.

I must also address a special word of gratitude to my friends and great companions of journey Tiago Gomes and Tiago Gomes, Sadro Pinto, and each and everyone who contributed to alleviate the burden of this quest.

My last and deepest gratitude is addressed to my fiancée Adriana, for her sincere love and support, confidence in my capabilities and cheering in the darkest moments; my father, my mother, my sister and close family for their affection and trust in me and in my life path; and to the cosmos.

This thesis was supported by a PhD scholarship from *Fundação para a Ciência e Tecnologia*, SFRH/BD/81681/2011.

Filipe Salgado

Guimarães, September, 2017.

"Quem quer passar além do Bojador
Tem que passar além da dor."

Fernando Pessoa

Abstract

Dynamic binary translation (DBT) has emerged as an execution engine that monitors, modifies and possibly optimizes running applications for specific purposes. DBT is deployed as an execution layer between the application binary and the operating system or host-machine, which creates opportunities for collecting runtime information. Initially, DBT supported binary-level compatibility, but based on the collected runtime information, it also became popular for code instrumentation, ISA-virtualization and dynamic-optimization purposes.

Building a DBT system brings many challenges, as it involves complex components integration and requires deep architectural level knowledge. Moreover, DBT incurs in significant overheads, mainly due to code decoding and translation, as well as execution along with general functionalities emulation. While initially conceived bearing in mind high-end architectures for performance demanding applications, such challenges become even more evident when directing DBT to embedded systems. The latter makes an effective deployment very challenging due to its complexity, tight constraints on memory, and limited performance and power. Legacy support and binary compatibility is a topic of relevant interest in such systems, due to their broad dissemination among industrial environments and wide utilization in sensing and monitoring processes, from yearly times, with considerable maintenance and replacement costs.

To address such issues, this thesis intents to contribute with a solution that leverages an optimized and accelerated dynamic binary translator targeting resource-constrained embedded systems while supporting legacy systems.

The developed work allows to: (1) evaluate the potential of DBT for legacy support purposes on the resource-constrained embedded systems; (2) achieve a configurable DBT architecture specialized for resource-constrained embedded systems; (3) address DBT translation, execution and emulation overheads through the combination of software and hardware; and (4) promote DBT utilization as a legacy support tool for the industry as a end-product.

Resumo

A tradução binária dinâmica (TBD) emergiu como um motor de execução que permite a modificação e possível optimização de código executável para um determinado propósito. A TBD é integrada nos sistemas como uma camada de execução entre o código binário executável e o sistema operativo ou a máquina hospedeira alvo, o que origina oportunidades de recolha de informação de execução.

A criação de um sistema de TBD traz consigo diversos desafios, uma vez que envolve a integração de componentes complexos e conhecimentos aprofundados das arquitecturas de processadores envolvidas. Ademais, a utilização de TBD gera diversos custos computacionais indirectos, maioritariamente devido à descodificação e tradução de código, bem como emulação de funcionalidades em geral. Considerando que a TBD foi inicialmente pensada para sistemas de gama alta, os desafios mencionados tornam-se ainda mais evidentes quando a mesma é aplicada em sistemas embebidos. Nesta área os limitados recursos de memória e os exigentes requisitos de desempenho e consumo energético,tornam uma implementação eficiente de TBD muito difícil de obter. Compatibilidade binária e suporte a código de legado são tópicos de interesse em sistemas embebidos, justificado pela ampla disseminação dos mesmos no meio industrial para tarefas de sensorização e monitorização ao longo dos tempos, reforçado pelos custos de manutenção adjacentes à sua utilização.

Para endereçar os desafios descritos, nesta tese propõe-se uma solução para potencializar a tradução binária dinâmica, optimizada e com aceleração, para suporte a código de legado em sistemas embebidos de baixa gama.

O trabalho permitiu (1) avaliar o potencial da TBD quando aplicada ao suporte a código de legado em sistemas embebidos de baixa gama; (2) a obtenção de uma arquitectura de TBD configurável e especializada para este tipo de sistemas; (3) reduzir os custos computacionais associados à tradução, execução e emulação, através do uso combinado de *software* e *hardware*; (4) e promover a utilização na industria de TBD como uma ferramenta de suporte a código de legado.

Contents

List of Figures

List of Tables

Listings

Acronyms

4LUT 4-input Look-Up Table

AAPCS Procedure Call Standard for the ARM Architecture

ABI application binary interface

AC Auxiliary Carry flag

ACC Accumulator

AES Advanced Encryption Standard

ALU arithmetic logic unit

AMBA Advanced Microcontroller Bus Architecture

APSR Application Status Register

BB Basic Block

BT Binary Translation

BCD binary-coded decimal

BEEBS Bristol Energy Efficiency Benchmark Suite

CAD computer-aided design

CAM content-addressable memory

CBD component-based development

CC Condition Codes

Ccache source code cache

CISC complex instruction set computer

COTS commercial off-the-shelf

CY Carry flag

DBT dynamic binary translation

DBTor dynamic binary translator

DSL domain-specific language

DSP digital signal processing

DDR double data rate memory

DFF D-type flip-flop

DIMM dual in-line memory module

DPTR Data Pointer

DWT Data Watchpoint and Trace unit

ECC elliptic curve cryptographic

EDA electronic design automation

EL Elaboration Language

EPSR Execution Program Status Register

eNVM embedded Non-Volatile Memory

ETM Embedded Trace Macrocell

FIC Fabric Interface Controller

FPB Flash Patch Breakpoint unit

FPGA field-programmable gate array

GCC GNU Compiler Collection

GP generative programming

GPIO general pupose input/output

GPL general-purpose language

GPU graphics processing unit

GUI graphic user interface

HDL hardware description language

IA Intel Architecture

IDE integrated development environment

ILP instruction-level parallelism

I2C Inter-Integrated Circuit

I/O input/output

IoT Internet of Things

IP intellectual property

IPSR Interrupt Program Status Register

IR intermediary representation

ISA instruction set architecture

ISR interrupt service routine

ITM Instrumentation Trace Macrocell

LE Lazy Evaluation

LR Link Register

LE logic elements

LSB least significant bit

MDD model driven development

MMU memory management unit

MSB most significant bit

MSS microcontroller subsystem

NRE non-recurring engineering

NRBG non-deterministic random bit generator

OO object-oriented

OS operating system

OV Overflow flag

P Parity

PC Program Counter

PSW Program Status Word

RAM random-access memory

RISC reduced instruction set computer

RTOS real-time operating system

SBT static binary translation

SCA Service Component Architecture

SPM Scratchpad Memory

SEU single event upset

SERDES Serializer and Deserializer

SFR Special Function Register

SHA Secure Hash Algorithm

SOA service oriented architecture

SoC system-on-chip

SP Stack Pointer

SPI Serial Peripheral Interface

SRAM static random-access memory

TBB translated Basic Block

TCM Tightly Coupled Memory

TPIU Trace Port Interface Unit

Tcache Translation cache

UART universal asynchronous receiver-transmitter

USART universal synchronous asynchronous receiver-transmitter

VLIW very long instruction word

Chapter 1

Introduction

In the present embedded systems' development trend, hardware is constantly evolving to meet the performance hungry market requirements. The commonly used method to increase performance can no longer be applied due to power constraints, but one approach is gathering consensus: the multi-core approach [2–5]. Multiple cores per chip increase the instruction level parallelism and by using specialized cores for specific tasks, rather than general processor cores. Therefore, the efficiency can be greatly improved, resulting in much lower power consumption [2,6]. Furthermore, the use of different cores simplifies the partitioning of different clock frequencies among the various system's cores in order to obtain large power savings and clock skew attenuation [6,7]. However multi-core architectures, tend to increase the price of the processor and, if not managed correctly, may have the opposite effect on the energy consumption. Thus multi-core approaches must be carefully evaluated on pair with the system's expected workload.

System variability is also hampering embedded systems development, due to the blooming of hardware and software integration possibilities. The ad-hoc integration of components in a system is no longer prudent once possible combinations arise exponentially. Moreover, functionality reuse and automatic code generation must be considered as a requirement, otherwise developers risk cost-ineffective production. To solve system variability concerns, corroborated software development techniques have being increasingly adopted. Techniques such as component-based development (CBD), model driven development (MDD), service oriented architecture (SOA) and generative programming (GP) are being applied in hardware development [8–11]. Applying CBD and MDD in full system deployment is justified by their efficiency in dealing with system modeling and functionalities description

and fusion. A design flow for hardware using MDD was proposed in [8, 9] taking advantage of system's meta-model specification. In [8], a high abstraction system level model using MDD model transformations was refined until the model reaches a detail level to automatically generate hardware description language (HDL) code. The model does not allow architecture exploration and performance analysis. In [9] a rapid prototyping tool (LavA) based on interconnected open-source tools was proposed. LavA successfully adopted software techniques to hardware, raising the abstraction level of hardware design. Similarly in [10], attention is drawn to MDD, using model transformation, concluding that there is a real need for techniques to model complex hardware systems.

Beside the mentioned fabrication process and development limitations, software and hardware legacy compatibility are also obstacles to advances in novel deployments. Legacy support for architectural compatibility must be assured, otherwise the novel hardware risks market rejection. Legacy support has multiple costs:

1. Hardware engineers must pay special attention, and spend valuable engineering effort to provide legacy support features to novel products;

2. These features might limit new products' performance due to backward compatibility;

3. Legacy support features may incur in significant runtime overhead.

On top of these constraints, embedded systems' design has tighter specifications than general purpose systems' design, due to smaller memory sizes, limited processing power and less resource abundant architectures. However these characteristics are the ones that mostly account to these system's reduced price, which is perhaps the main aspect that leads to their broad adoption in the industry's sensing and monitoring and control applications, home automation, small appliances, mobile devices, technology gadgets, etc. Even in the embedded systems category, there is a great variety of devices, with different architectures, processing power, power consumption and memory resources, among other characteristics. Here, also the lower resourced systems will have a smaller cost than the well endowed ones. Therefore, embedded systems may also be categorized mainly as low-end and high-end. The separation boundary is not well defined, but a low-end or resource-constrained embedded systems typically have a single core, a clock frequency up to 100 MHz, memory resources up to 128 KB random-access memory (RAM) and 256 KB flash and by last, should cost well bellow the $10 price mark [12]. These type of systems

are well disseminated as commercial off-the-shelf (COTS) solutions, thus its usage is leveraged from immediate commercial availability, ready-to-use development platforms and vast support and tool-chains, resulting in faster time-to-market.

It is precisely the low-end embedded systems that proliferate in the industry since early times, due to their broad dissemination and wide utilization in sensing and monitoring processes. Many of these systems are still correctly working and under use at the current days, however, there are risks and costs that arise with such utilization:

1. These systems have higher maintenance costs due to obsolete components utilization;
2. A system failure may results in heavy production loss since there are no replacement parts;
3. Legacy systems tend to be very energy inefficient;
4. They were not designed with security as a concern.

The re-design of these embedded systems is often costly due to several causes:

5. The source code of the firmware does not exist or was lost, therefore recompilation of the code to modern systems is not possible;
6. A modern replacement system involves the design from scratch of the whole solution, including requirements gathering, non-recurring engineering (NRE) development costs, testing and sometimes certification costs of complex projects.

Hence, the possibility of using the legacy systems binaries into a contemporary replacement platform is highly attractive because not only prevents the events 1., 2. and 3., but also may address the challenges of 4. and alleviates the costs of 5. and 6. with the added benefit of the low energy consumption associated with modern architectures. This motivates the migration of dynamic binary translation (DBT) techniques to embedded systems for legacy support purposes, most specifically targeting the low-end architectures due to their reduced price, high versatility, broad dissemination and contained energy consumption.

Dynamic Binary Translation

Binary Translation (BT) is a technique that was developed for architectural compatibility, i.e., to run machine binary code on architectures different from the one it was compiled for (Figure 1.1). This technique also eases the bridging of legacy systems to cheaper and up-to-date platforms [13], providing binary compatibility with minimal NRE.

There are several ways to run binary code in a different machine for which it was produced. The most common ones are using an interpreter (emulator) or a BT [14, 15]. While the former re-creates a source machine architecture model in software and interprets instructions, the latter translates the binary files and creates a new machine code to be directly executed in the new target. BT is for a long time widely accepted as a better solution than pure interpretation due to performance reasons, from between 5× up to 10× faster [16, 17], and code optimization opportunities created during translation process [17–19].

Dynamic over Static BT

There are two approaches to BT, namely static or dynamic. HP labs pioneered the static binary translation (SBT), by porting emulation techniques and combining a machine emulator and a code translator [20] . The translators became quite common in the '90s, when hardware manufacturers provided translators to encompass the migration from their complex instruction set computer (CISC) instruction set architecture (ISA) to reduced instruction set computer (RISC) ISA [20–25]. Static binary translators follow the structure of a compiler, having a front-end or a de-

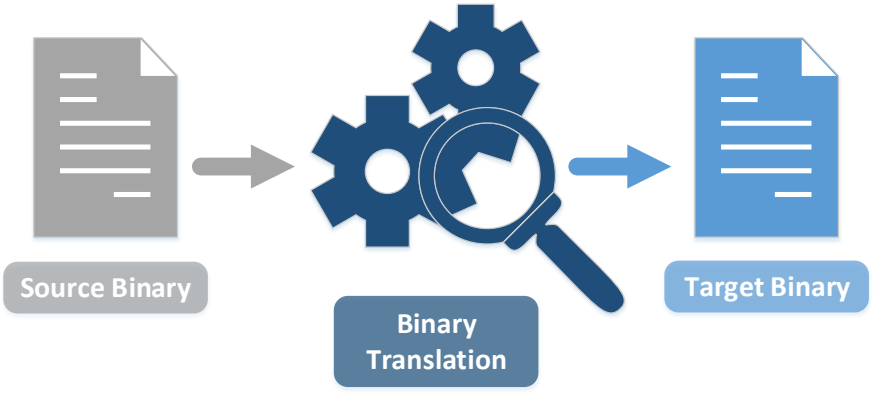


Figure 1.1: Binary translation general concept.

code stage, an analysis and optimization stage and a code generation or back-end stage. SBT works well for bare metal programs or user space programs in systems with an operating system (OS). These kinds of binaries usually have well separated code and data memory spaces, and do not use programming techniques such as self-modifying or dynamic code generation, which makes them very suitable for SBT [15]. In addition to self-modifying or dynamic code generation, SBT can not deal efficiently with indirect branches and indexed memory accesses because these behaviors rely on computations performed during run time, which are not known statically. These situations require a fallback mechanism, which emulates the runtime environment and interprets untranslated code. When compared to SBT, DBT can solve many of the SBT problems and drawbacks: (1) code discovery issues, hard to handle in SBT [26, 27], do not exist in DBT since executable code is identified during runtime program flow; (2) self modifying code can be handled during runtime, by invalidating cached code and generating target machine code again [14, 15]; (3) indirect jumps and indexed address modes are calculated and translated during runtime [15]; (4) DBT is exempted of the runtime environment development cost [14]; (5) DBT easily deals with non-user space applications, which means that full systems (OS plus user space binaries) can be translated to another target architecture [28] by integrating virtualization support in the ISA translation layer, opening a new path to system virtualization.

DBT technology has been used to offer several other services than binary code compatibility, such as, co-design innovative micro-architectures, code security, power management, software caching, program analysis and instrumentation, ISA virtualization support, among others. Conceptually, its core consists of a runtime engine and a translator which fetches guest application code, decodes and transforms it according to the provided service (e.g., cross ISA compatibility, virtualization, optimization), while caching the translated code in a translation cache, from where it executes directly on the target machine. Therefore, DBT technology incurs in significant overhead which degrades performance due to decoding and translation, optimization, emulation and other potential runtime overheads. Furthermore, the implementation effort is non-trivial as porting a DBT layer, which is typically architecture and service-oriented, to a new system, requires considerable engineering effort and may incur in unacceptable performance penalties.

DBT for Virtualization

Examples of DBT for virtualization purposes are presented in [29–35]. In [32] it is presented a Linux-based DBT virtualization layer which translates memory pages demanded by an application through the use of a co-processor. In [29], DBT is used by VMWare to translate non-virtualizable instructions. In DBTIM [30, 36] the authors provided alternative solutions to paravirtualization in architectures that were not meant to be virtualized, using reconfigurable hardware (FPGA) to perform the translations process and replace non-virtualizable instructions. Modern days hypervisors [34,35], still utilize DBT for full-system and application-level virtualization on high-end embedded ARM platforms.

DBT for Binary Compatibility

Different reasons, from different interested parties, to invest in legacy support and good binaries migration solutions are exposed in [14]. They are either organizations that want to maximize investments in specially developed software, to take advantage of their full capabilities; or hardware designers, who develop new architectures and want programs migration to run with full performance; or even software developers, who want to guarantee correct functionality of programs for a variety of platforms with reduced testing time. FX!32 [16], DAISY [37, 38], Crusoe [39–41], BOA [42–44] and Harmonia [45] are all examples of binary compatibility investigation, however not all the discoveries made so far have been made public, due to commercial interests from the involved parties [14]. The initial trend was to support a competitor's ISA on a hardware's developers own architectures [16,37–44] for market acceptance reasons [16, 39] and to benefit from the advantages of very long instruction word (VLIW) architectures such as design simplicity, high instruction issue rate and high instruction-level parallelism (ILP) [37, 39, 42]. DAISY [37, 38], is one of the pioneer works in DBT for VLIW target architectures. The work addresses several challenges in DBT, such as, page and address mapping, cross ISA Condition Codes (CC) emulation, translation cache management, exception scheduling, self-modifying code and aggressive reordering of memory references. Transmeta attempted the same targeting low-cost, high-performance proprietary VLIW microprocessors with Crusoe [39]. More recently, Harmonia [45] resourced to DBT to provide compatibility of embedded market associated architectures with Intel Architecture (IA), motivated by the today's ubiquitous presence of ARM plat-

forms. Two challenges were identified: the reduced number of registers in the IA architecture and the condition codes handling divergence in both architectures.

DBT for Code Optimization

DBT is also used to accelerate system simulation as shown in [46], where the authors demonstrate the use of homogeneous multi-processor system to perform in parallel the translation, optimization and execution of several hot traces. Examples of DBT application in code instrumentation and dynamic optimization are presented in [47–54]. In DBT, dynamic optimization is tightly related with code instrumentation because the optimization actions often result from information gathered during code runtime instrumentation. Thus the heavy optimization algorithms are applied only where hot code traces are detected. Strata [47, 48] and HDTrans [51, 52] are two popular DBT frameworks for this purpose, which may be extended for specific services (e.g., ISA simulation, instrumentation, OS emulation, optimization). The versatility of the STRATA was explored in plenty of works based on built around Strata project [55–61]. In fastBT [53, 54], Payer and Gross propose optimizations to reduce the DBT runtime overheads observed in DynamoRIO [49], HDTrans [51] and PIN [62]. FastBT achieves a low overhead of 0% to 10% based on a fast mapping lookup of the trace cache and configurable in-line optimization mechanisms. These DBT systems however do not perform cross-ISA translation, since their use is more like a performance monitoring and development tool, which discourages cross-ISA compatibility application purposes.

DBT to Reduce Energy Consumption

In [63], the authors show how runtime-profiling can be used to perform efficient frequency and voltage scaling, successfully decreasing the power consumption up to 70%. Other power management techniques are proposed in [64], where runtime optimization is used to perform register re-allocation, thus decrease cache accesses and so, reducing the energy consumption. In [65], Schranzhofer *et al.* proposed a method to minimize the average power consumption by intelligently map tasks to key processing units. The presented method adapts to the current execution pattern. Hong *et al.* in [66] directly relate the energy consumption with the performance and number of instructions executed. In DBT systems, if the translated code is far more extensive that the original, it means that not only performance,

but also energy consumption will suffer a negative impact. Focusing on mobile applications, the authors transferred the heavy translation/optimization workload to a fixed and power unconstrained server, which upon request, receives the IR of hot traces of code, performs aggressive optimizations and sends the result back to the thin client, which has a light translator/emulator running. A remote thin client was also used in [67, 68] to reduce the DBT negative effects on memory of resource-constrained systems.

Overheads Mitigation Methods

The main drawback of traditional DBT techniques [48, 69] is the performance overhead caused by translation and emulation [70]. This is primarily due to basic emulation algorithms, such as identified in [69], and the difficulty in identifying proper optimization opportunities. Furthermore, optimizations during runtime are a heavy burden, constrained by the amount of time that the system can spend in such process without performance impact [71]. In [72, 73], the authors faced the high overhead of DBT by combining a dynamic binary translator with a previous static analysis stage of the code to be translated. With the profile information collected statically, some optimization algorithms are run before load time. This approach eliminates the profiling and heavy optimization overheads, resulting in a runtime reduction of more than 34% with a memory increase cost of just 2%.

Dedicated hardware and parallel execution are other approaches to DBT overhead mitigation. Dedicated hardware provides additional processing capabilities to deal with DBT specific tasks without using the existent system processing power. CoDBT [74] attempts to mitigate the performance overhead by migrating DBT functionalities to hardware, without making changes to the target processor, successfully implementing parallelism. GODSON [75] translates x86 binaries to its target MIPS architecture through dedicated hardware. The Code Morphing technology implemented in Crusoe [40] translates x86 through a hardware/software mix. Alternative DBT uses, such as Warp processing [76], have been used to increase the optimized code performance, but suffer from significant overheads caused by FPGA reconfiguration. Task parallelization is another efficient way to reduce overhead and increase performance. This is a buzz area in research since one of the modern era uses of DBT is the parallelization of code to increase ILP. Sokolov *et al.* [77] attempted a software-only solution by performing optimization in parallel with execution, on a different software thread, running on the

same or other core. MTCrossbit [78] and MT-Btrimer [79] use multi-thread execution to address context switching overhead mitigation by mapping translation and executions tasks into different software threads [78]. MT-Btrimer also performs speculative code translation on a master-slave DBT architecture, achieving on average 30% performance improvement at heavy Translation cache (Tcache) burdening cost. In [78], a solution that targets multi-core processors demonstrates how running each DBT functionality in a different core can contribute to increase performance. In [17, 19] the authors try to make full use of multi-core platforms and their performance by parallelizing different DBT tasks like code translation, code optimization, code execution or speculative translation direction estimation. BOA [42–44] also demonstrates the use of DBT to increase performance and processing speed by using a dedicated processor core for translation.

Legal Considerations

As a final consideration on DBT, the legality of the platform emulation process is also seen as a barrier to its adoption. The legality issues are split in two aspects: one is related with the emulation of the system and the other is related with the copy of the software to be emulated [15]. Regarding the creation of an emulator, detailed information must be known about the original architecture and most of the times that information is copyrighted. If that information is stolen, or obtained without the consent of the owner, the creator of the emulator may be sewed and the emulator can never be sold legally. Another way to obtain that information is by reverse engineering, and that technique remains in a gray legal area. Regarding the software copies, the duplicates of the binaries are illegal unless the owner also owns an original copy of that software, which in industry is a mandatory practice, but still a delicate topic. In DBT, in fact no software copies are made, since only a translation of the original software is stored temporarily at volatile memory, thus no copies persist.

1.1 Scope

This thesis is focused on binary compatibility in embedded systems through DBT techniques. This motivation comes from the need to provide legacy support for embedded systems binaries at a reduced cost, using COTS products and as a

reusable solution, applicable to multiple scenarios. System-level type DBT will be addressed since most of the low-end industrial legacy systems would likely run bare-metal single task applications, thus the use of an OS can be discarded. From the presented general overview on DBT and embedded systems, certain features and characteristics are very favorable to contribute to this thesis goal, such as the use of COTS embedded devices, cross-ISA translation and hardware acceleration. The use of COTS devices contributes to the low price of the final solution and also favors its deployment due to the availability of ready-to-use development platforms. Cross-ISA translation is crucial to support legacy binaries and, allied to an intermediary representation (IR) between source and target ISAs, will favor the portability of the solution to multiple application use cases. Regarding hardware acceleration, it is an attractive technique for overhead mitigation and parallel optimization with little to no impact on the DBT resource allocation, that applied in embedded systems might solve many performance issues. Reconfigurable hardware is also becoming a common feature in today's COTS embedded SoC, which is a opportunity that should be explored. The solution must also be integrated on a framework that may provide a higher abstraction level model for configuration and customization, and that enables automatic code generation and solution space exploration.

Thus the solution space of this work is configured between (1) a legacy support

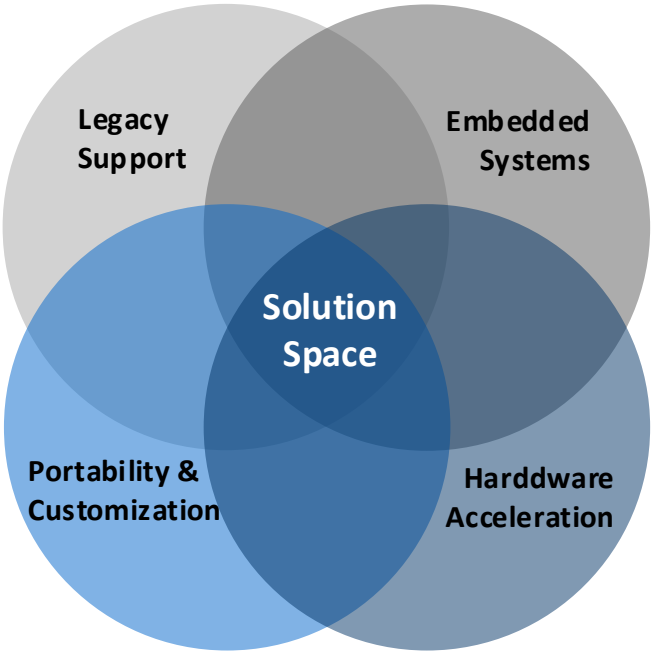


Figure 1.2: Solution space.

at binary level, (2) low-end COTS embedded systems, (3) ease of customization, resourceability and retargetability for different application and (4) acceleration/optimization through hardware, as illustrated in Figure 1.2.

A deeper analysis on DBT resourceability and retargetability through IR, hardware assisted DBT and automation enablement solution are presented ahead.

Despite most of the optimizations carry out in DBT system being to increase performance, when considering an embedded environment, energy constraints should also be highly considered as an optimization goal. However, this thesis does not directly addresses such issues due to scope limitations. Other relevant and interesting concerns that are also kept away from the scope are real time requirements, application level DBT and ILP exploration either by multi-thread or multi-core approaches.

Resourceable and Retargetable DBT through IR

A DBT engine which performs runtime optimization requires a structure similar to a compiler. It has a front-end or a decode stage, that produces an IR of the source binaries; a middle-end where code analyses and optimizations are performed over the IR; and a back-end stage, where the target code is generated [80].

For resourceable and/or retargetable translators, i.e., translators that use the same intermediate structure to translate code from multiple source architectures to multiple target machines, the use of the IR brings additional advantages, since the IR acts like an abstraction layer between the source and target ISAs, decoupling the code decoding and generation stages [81]. In practical terms, the decoding stage of an IR resorting DBT engine must decompose the source instructions into IR sub instructions, and the code generator must generate target code for each of the IR's sub instructions. This approach separates the source and target specific components of the translator from the DBT kernel, and introduces different source or target architectures support without considering the counterpart architecture, since the porting must address the IR only.

DesIRer [81] is a multi-platform binary translator which applied the conversion of the GNU Compiler Collection (GCC) into a retargetable binary translator. The IR and back-end of the compiler is inherited by the developed translator for multi-target support as an economic way for obtaining retargetability in binary translation. For an x86 to ARM translation, performance degradation was ob-

served due to translation overheads from load and store instructions introduced by the CISC to RISC translations.

Bintrans [82–84] is a machine adaptable binary translator, at application-level, for Linux. The work is a cross-ISA translator developed for the purpose of DBT research and supporting translation from source x86 and PowerPC to Alpha and PowerPC target architectures. The authors also achieve resourceability and retargetability by using and IR for front-end and back-end separation, and identify the disadvantages of such approach, namely: (1) translation through IR is slower than translating directly from source to target code, as also identified in [85]; and (2) it is hard to obtain an IR which can capture every ISA's architectural details, (e.g., CC), and the broader the IR the less efficient target code generation is.

Bellard developed QEMU [86], a popular portable machine emulator that uses DBT for optimization of frequently executed code blocks. QEMU supports full-system emulation and user-level emulation for Linux OS. The multiple architecture support is implemented through an IR which uses *micro operations* to represent the source instructions into target architecture binary code. The vast dissemination and popularity of QEMU is due to its great portability and architectures support, achieved though the use of the IR. Originally the QEMU IR was ported through "dyngen", using *micro operations* object file information as input to generate target machine code. QEMU is a desktop application, thus uses large translation caches (16 MB) to accommodate the translated code and performs simple optimizations such as lazy evaluation of the CC and direct block chaining.

In Hermes [87], the use of an IR is identified as severe source of performance penalty, preventing effective ISA-specific optimizations. Hermes addresses this by eliminating the ISA and performing post-optimization on the generated code and not on the IR, as commonly performed. The authors claim that the technique used, Host-specific Data Dependence Graph, compared to QEMU, is able to achieve performance speed-up up to 3.14× and 5.18× for x86 and ARM guest ISAs, respectively, preserving the portability offered by and IR. However this approach was deployed on a superscalar processor [75], thus its use in resource-constrained embedded systems is highly discouraged.

CrossBit [88] is a multi-source multi-target cross-ISA DBT. Like QEMU [86], it uses an intermediate representation, VInst, between source and target instructions as an abstraction layer in order to ease the multi source and target portability. CrossBit was able to outperform QEMU in several benchmarks due to several

optimization efforts: hot code optimization into code super-blocks, translated code blocks chaining and indirect jump inlining. The VInst design is based on some characteristics that grant its efficiency at targeting low level machine instructions:

- It is regular and simple, for good low level instruction matching;
- Similar to a RISC ISA for versatility and simplicity also;
- It has an unlimited numbers of 32-bit virtual registers;
- "Load-Store" style architecture, meaning that only 'load' or 'store' instruction can access the memory;
- Base plus displacement is the only addressing mode;

The authors highlight the impact of a well designed IR in the quality of generated target code, and therefore in the translator performance, and advise a balance between the performance and the cost. That is, the IR should be kept simple to reduce the cost of translation. However it should have a semantic rich enough support to various characteristics of different ISAs.

Hardware Assisted DBT

The use of hardware to assist software processes is not new and is well disseminated and proved in achieving considerable performance improvements [89–92], through (1) accelerating iterative software tasks, (2) task parallelization and (3) improving predictability and determinism. However, its application in DBT is not very common, mainly because software acceleration techniques are used as a first option, since the major target systems of DBT are high-end and well resourced systems. In embedded systems, the resources are well contained and limited almost to the application's essentials, thus hardware assistance may have a different role in these systems.

In DBTIM [30], hardware assisted DBT (in FPGA) is used to improve a virtualization solution without modifications in the target ARM processor. The DBT systems is deployed on a dual in-line memory module (DIMM) and then connected to the target board like a regular memory. The solution is COTS compliant, however requires two additional chips, the DBT execution core and the FPGA, which increases the cost of the solution. Nevertheless, it is non-intrusive capable on what concerns execution parallelism, hardware compatibility and flexibility. Later, the

same authors have proposed efficient mechanism to reduce interpretation overhead by mean of hardware assisted interpreted code cache and a decoded instruction cache, with promising simulation results [31, 33].

Warp processing [76] have been used to increase the optimized code performance. This technique consists in dynamically profiling the execution of binaries in order to detect the hot code traces. Those traces are synthesized and mapped to FPGA logic by dynamic computer-aided design (CAD) tools and the original binaries are modified in order to use the new circuits. The whole approach is computation- and resource-heavy but also results in great performance speedups (up to 169×), specially in tests with a well defined execution kernel. The technique is however not suitable to cross-ISA translation, but proves the suitability of FPGA and DBT integration.

CoDBT [74], from the same authors of CrossBit [88] , is a multi-source to PowerPC collaborative hardware/software co-design version of the CrossBit. The authors tackle Tcache management and translation overheads with reconfigurable hardware modules deployed in FPGA, while proposing a mixed mechanism to collect program behavior information with software-based instrumentation and hardware-based profiling.

In [93] the shortcomings of dynamic binary translation are tackled by Dynamic Reconfiguration. Through partial reconfiguration, FPGA fabric reserved to translation services can be redirected to house execution services. If reconfiguration is done in useful time, accumulated workload can be dispatched quicker, reducing execution time, thus saving power. However, the very promising work seems to never have passed initial simulations.

Automation Enablement

DBT utilization as an end-product in the industry has been hampered by the complexity of the subject and its associated variability management, which brings configuration challenges to the final solution [94]. The accessible and profitable use of DBT requires design automation paradigms and variability management solutions, expanding its usage for DBT laymen.

In [95], Kondoh and Komatsu propose a specialization framework to generated host code to exploit a limited number of characteristics (memory management unit (MMU), bi-endianess and register banks). Their contribution however do not

offer the necessary flexibility to be applied on different translators or to support other configurations than the ones provided. To achieve such type of tool, a robust base system model is required. This kind of model must comprise not only the components of the translator and their characterization, but also specify interfacing rules and foreknow the accepted variations of the model on a higher abstraction level, that is subsequently lowered until reaching the implementation code files and the executable binaries.

LLDSAL [96] and PACT:U [97] are two domain-specific language (DSL)s applied to DBT for automatic translation generation. Despite both approaches applying DSL in DBT, the modeling of the DBT architecture and the generation of translation for mismatching source/target pairs are not supported. This is due to their application scope, which is application security [96] and code instrumentation [97].

1.2 Research Questions and Methodology

As an endeavor to approach legacy support on resource-constrained embedded systems through DBT, while bearing in mind the expected overhead worsening of these systems, this thesis pursues the answer to the following research question:

How to leverage an optimized and accelerated dynamic binary translator, targeting resource-constrained embedded systems for legacy support?

Such broad question must be divided into smaller and detailed sub-questions, which are presented ahead:

1. Is DBT a possible solution to address the legacy support challenges on the resource-constrained embedded systems, allowing the use of modern low-cost and low-power architectures?

2. How to attain a flexible, yet application-tailorable solution, while minimizing NRE development costs and maximizing the solution applicability?

3. How to address the overheads associated with DBT, considering the low resources of the target devices, while providing full legacy support?

4. How to manage system variability and enable solution space exploration and automation, or putting in other words, how to promote DBT utilization as

a legacy support tool for the industry as an end-product?

To evaluate these questions, the following methodology is proposed:

1. *Design a specialized DBT engine for the resource-constrained embedded devices in order to evaluate the feasibility of the solution*, identifying the base components of a DBT engine and bearing in mind the limitations associated with such type of target systems. Evaluate the performance of the obtained solution and identify overhead sources for optimization/acceleration exploration.

2. *Include customization features for reconfiguring the solution and deciding for a multi-source and multi-target architecture.* Adopt an object-oriented (OO) programming in the development for better code reuse and recycling, reducing the NRE efforts in the solution tuning.

3. *Explore a hybrid DBT architecture solution by resourcing available hardware for acceleration support* in COTS target devices. Investigate the possibilities and limitations of such approach for DBT functionality extensions.

4. *Develop a model-driven domain-specific language for DBT architectures and a framework for ready-to-use DBT solutions*, to enable design space exploration and ready-to-use application tailored DBT solutions.

1.3 State of the Art

This section presents the existing literature on DBT for embedded systems, reviewing the existent DBT engines which somehow target embedded systems, even if with different purposes than legacy support.

Kondoh and Komatsu have developed work on the specialization of DBT for embedded systems simulation in [95], proposing a novel technique for enhancements at the MMU, endianess and register banks. The specializations produce execution code up to 30% faster. The authors take a different approach to DBT and explore one key assumption of embedded systems: systems intended to perform one or a few dedicated functions. They explore this by representing the source architecture as a machine state, which is modified by the source instructions, with the translator acting as a state machine-aware controller. The work contemplates resourceable capabilities, and mention the use of and IR, but without further de-

tails. The authors include interrupts and peripherals support in their approach. The work however is not a legacy support binary translator, but a simulator of embedded systems architectures targeting the powerful Intel Xeon cores.

Baiocchi *et al.* [61,98] developed extensive work in DBT for embedded systems, addressing the problems and overheads found in the literature through optimization and enhancement techniques applied at the translation cache. The work StrataX is based on the Strata DBT instrumentation engine and does not cope with cross-ISA translation. The authors identify the negative effects of a bounded Tcache in DBT, typically found in embedded systems due to memory constrains, and propose mechanisms to alleviate such effects through: (1) reduced control code portion of the translations, and (2) Tcache management and optimization techniques. Despite studding the extra challenges that embedded systems add to DBT, from this thesis's scope perspective, the work has several flaws: (1) it is fundamentally a Strata porting to embedded systems, thus directed to code instrumentation; (2) it considers fairly resourced embedded system targets, a virtual 64-bit architecture (PISA), uncommon in the low- and mid-end embedded systems; (3) it is purely based on simulation results from the SimpleScalar; (4) and requires the existence of a Scratchpad Memory (SPM) or tightly-coupled/on-chip memory.

In [99] is presented a hardware/software co-designed DBT system, supported by FPGA fabric for functionality offloading and acceleration in cross-ISA compatibility DBT. The authors propose small ISA extensions and full CC hardware emulation, together with auxiliary hardware address mapping for up to 56.1% performance improvement. The work however implies architectural modifications on the processor, thus it is not COTS compatible. No information is provided regarding the type of translator implemented, the use of IR nor the emulation of peripherals and exceptions. The work however contributes with good hardware acceleration techniques to the state of the art.

Richie and Ross achieved cycle accurate support of 8080 legacy binaries on the deprecated hi-end ARM11 architecture in [100]. Despite the 350-to-1 target/-source clock ratio (700 MHz clock target system) the authors evolved an existent emulator to support BT and implemented optimization features such as a direct jump table for instruction decoding, direct source/target register mapping, and no intermediate representation. The approach proves the practicality of DBT use for legacy support in the resource-constrained embedded systems without negative performance effects on the legacy program's execution. However, the portability of

the solution to other source and target architectures was not approached, neither the peripherals emulation support.

Gap Analysis

From the state of the art, and considering the scope of this thesis, an evaluation was accessed considering the following parameters:

- **Embedded system's range** of the target device, indicating for what type of embedded system the translator is targeted at: low-end or high-end;
- **Application purpose** of the translator, *i.e.,* the suitability of the translator for legacy support application;
- **Cross-ISA translation**, *i.e.,* the capability of the translator to generate code for a different ISA than the original;
- **Resourceable** refers to the capacity of the DBT engine to support multi source ISAs, at a low engineering effort;
- **Retargetable** refers to the capacity of the DBT engine to support multi target ISAs, at a low engineering effort;
- **Hardware acceleration** indicates if any type of acceleration by hardware was introduced in the translator;
- **COTS compliant** evaluates if the DBT engine may be hosted or ported to standard devices or if was developed considering the utilization of commercial closed architectures;
- **Peripherals emulation** is the evaluation of the peripherals (and interrupts) support provided by the translator;
- **Customization framework** indicates if any framework or customization tool was presented for easy configuration of the solution.

The findings of the assessment are presented in Table 1.1. From there, it is possible to conclude that the state of the art fails in providing answers to the presented research question, i.e., none of the existing DBT engines offers all the parameters considered to be necessary for a complete DBT tool for embedded systems legacy support, as well as none of the systems contemplate resourceability and re-

Table 1.1: Gap analysis between existing DBT engines.

	Kondoh & Komatsu [95]	***Baiocchi et al.*** [98]	***Yao et al.*** [99]	***Richie & Ross*** [100]
Embedded systems' range	n/a	high-end	low-end	high-end
Application purpose	systems simulation	code instrumentation	binary compatibility	legacy support
Cross-ISA translation	yes	no	yes	yes
Resourceable	yes	no (PISA)	no (x86)	no (Intel 8080)
Retargetable	no (Intel Xeon)	no (PISA)	no (MIPS)	no (ARM11)
Hardware acceleration	no	no	yes	no
COTS compliant	yes	yes	no	yes
Peripherals emulation	yes	no	no	no
Customization framework	yes	no	no	no

targetability requirements. The work from Baiocchi *et al.* [98] does not even offer cross-ISA translation. The solution proposed by Kondoh and Komatsu [95] despite offering a customization framework and fulfilling other requirements is directed to systems simulations, thus is not eligible for binary legacy support purposes. The remaining works [99, 100], despite eligible to legacy support DBT, fail in offering peripherals emulation features, a customization framework and providing both hardware acceleration possibilities and being COTS compliant deployments. The state of the art is fitted in the presented solution space diagram in Figure 1.3.

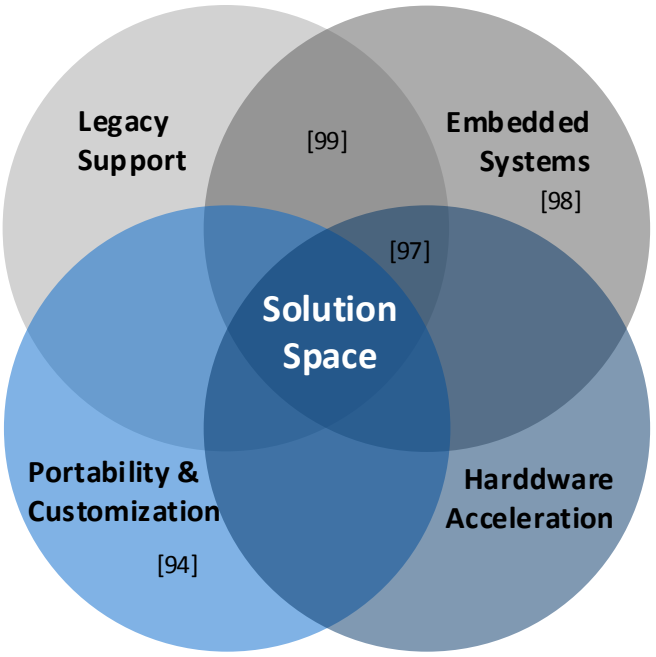


Figure 1.3: State of the art placement in the solution space.

1.4 Thesis Structure

The content of this thesis is organized in eight chapters, presented as:

- **Chapter 1**, the present chapter, is where DBT is introduced with its application purposes and challenges as well as the motivations for its application in embedded systems, accordingly with the bibliography. The scope of the work is defined and the research questions are drawn, followed by the used methodology and the statement of the contributions.

- **Chapter 2** presents the research tools and materials used in this work. A brief discussion of resource-constrained legacy and modern low-end embedded architectures is conducted, leading to the selection of one source and one target architectures for source/target study pair in the DBT engine to be implemented. The development platform and the evaluation benchmarks are also presented in this chapter.

- **Chapter 3** introduces a generic resourceable and retargetable dynamic binary translator (DBTor) deployment for the resource-constrained embedded systems, addressing the customization and code reuse concerns expressed. The underlying IR is presented, together with the architectural components and other relevant functional mechanisms.

- **Chapter 4** is the follow-up of Chapter 3, where the DBT demonstrator is realized, integrating the obtained generic DBT engine with the selected source and target embedded architectures from Chapter 2, in order to reproduce a legacy support case scenario. The source and target porting tasks are presented separately, as well as the CC emulation mechanism. The performance of the system is evaluated through the execution of 10 benchmarks and the overhead sources, candidates for optimization/acceleration are identified.

- **Chapter 5** evaluates different CC emulation mechanisms, in an attempt to tackle the associated emulation overhead identified in the previous chapter. A novel approach is suggested and evaluated, using the debug monitor hardware as a CC lazy evaluation trigger.

- **Chapter 6** introduces several reconfigurable hardware based components into the translator, contribution to a software/hardware hybrid DBT solution. Tcache performance improvements are presented with a partially hardware deployed solution; and a hardware sniffer based architecture, for functionality acceleration and extension, is also proposed.

- **Chapter 7** presents a component-based DSL for DBT modeling and wraps the DBT contributions proposed through the thesis on a framework, pursuing a ready-to-use and automation enabling DBT tool for resource-constrained embedded devices.

- **Chapter 8** is where the final conclusions of the work are drawn, discussing the limitations of the work and suggesting future research paths on the matter.

1.5 Conclusions

This chapter throughly introduces the thematics of embedded systems design and DBT, addressing used techniques and challenges, accordingly to the existing literature. The motivation of the thesis is expressed and the associated research questions are presented, followed by the proposed research methodology. The conducted state of the art analysis shows that currently no solution provides answers to the posted research questions, exposing an investigation gap on DBT for legacy support in embedded systems. By last, the organization and content of this thesis' chapters is presented and briefly described.

Chapter 2

Research Tools and Materials

In this chapter the research tools and materials used throughout this thesis are introduced. In section 2.1 the potential source and target architectures to configure the thesis' DBT demonstrator are discussed and selected. In Section 2.2 and Section 2.3 are addressed the relevant characteristics of the selected source and target architectures, respectively. Section 2.4 introduces the research and development platform used and in Section 2.5 the tool used for benchmarking is presented and asserted.

2.1 Source and Target Architectures

Despite driving the research effort towards an architectural agnostic solution, in order to experiment and test the proposed solution, it is necessary to implement a demonstrator. To do so, one source and one target architectures must be selected. This source/target pair should replicate as possible application scenario of the translator, so appropriate ISAs should be selected accordingly to the established application criteria in the section 1.1. The selection criteria for both ISAs was made considering the legacy support purpose for resource-constrained embedded devices and the separated source/target DBT architecture.

The source ISA must fulfill the following characteristics:

- Legacy ISA, meaning that the processor's architecture was common and widely used by previous generation of embedded systems in many home appliance and industrial applications. Despite possibly still existing, the

processors might be out of date in features or in system compatibility and the replacement is usually hazardous:

  - Non-compliance with moder industry standards and limited Internet of Things (IoT)-integration capabilities;
  - Lack of necessary cyber-security features;
  - The costs associated with the acquisition of replacement hardware;
  - The re-design of the system is a complex and expensive task;
  - There is no software documentation or available source code.

- Used in resource-constrained embedded systems, meaning that the ISA range of applications should not span to desktop and other high performance applications.
- Simple ISA, without complex instructions. This criteria is applied to simplify the porting effort because it is not the focus of the work.

On the other hand, these are the desired features of the target ISA:

- Von-Newman architecture or a modified Harvard architecture, allowing write-access to the program memory, for code generation.
- A modern ISA, widely spread among the embedded systems arena, with broadly available tool-chains.
- A low-cost but efficient architecture currently used in low-end embedded applications.
- Support and continuity, meaning that it is very unlikely that its use become deprecated.
- Robust debug support for software analysis and problem detection.
- Low energy consumption is an extra feature, because it is an appreciated characteristic, however energy costs studies are outside the scope of this thesis.

Based in these criteria, some source and target ISAs candidates were gathered and studied, and therefore are presented and discussed in the forthcoming paragraphs.

2.1.1 Source Candidates

When discussing legacy support, one of the first architectures that relates with the topic is the Intel MCS-51, a.k.a. 8051 [101]. This early 80's 8-bit architecture was very common at that time and still is very popular nowadays. Due to its versatility it is broadly disseminated in home appliances, automotive systems and data transfer and user interfaces. Because of its success there is a plethora of different versions of the main architecture, either with performance improvement, enhanced peripherals, reduced energy consumption, etc. These variations lead to different peripheral mappings, different memory organizations and slightly different instructions effects, leads to legacy compatibility problems. This is mainly due to the fact that Intel released the silicon intellectual property of the microcontroller's family.

The PIC is another famous 8-bit architecture from the early 80's [102], owned by Microchip, which also integrates the potential source ISAs list. Its huge popularity was due to the very low cost, simple RISC ISA, free development tool-chain and fast application deployment. Originally developed in 1975 as a "Peripheral Interface Controller" - (PIC), its simple execution model made it last until modern days. Its applicability was expanded besides the input/output (I/O) interface and currently there are several PIC device families, varying the number of available general pupose input/output (GPIO)s, memory space, architectural resources and registers and instructions wideness. There is no binary equivalence among the instructions of the different families and devices. Due to these characteristics this architecture was selected as good source ISA candidate.

The Motorola 68HC11 [103] and the Zilog Z80 [104] were also analyzed for being MCS51 and PIC's contemporary 8-bit architectures and because they share some common characteristics. These architectures were also both aimed at embedded systems and popular three decades ago. The 68HC11 was used in automotive, amateur robotics and small embedded devices, while the Z80 was designed for higher-end products, like advanced register machines, industrial robots and automation, video-games machines, music synthesizers and small home computers. The Zilog architecture got a lot of attention due to the success of one of the products incorporating it, the ZX Spectrum.

After a careful analysis of all of the architectures, the Intel MCS-51 was selected. Despite the Z80 being a very good candidate for the porting, it was a higher-end

products and its complex ISA could introduce additional undesired complexity to the project. Between the MCS-51 and the PIC the choice were towards the Intel architecture because it is an open source architecture, as well as due to the author and supervisors experience with the architecture, a valuable valence in the porting task.

2.1.2 Target Candidates

On the target side, multiple candidates to host the DBTor were identified. Several architectures were studied, namely the ARMv7-M (Cortex-M3), the Atmel AVR and the MIPS architectures. They all fulfill the initial premises: a popular present-day architecture, with good tool-chain support and low-cost, low-energy consumption.

ARM has several different core architectures in the market, all dedicated to the embedded systems, however, to meet the constrains of this project, the selection was centered on a mainstream cost-effective core, widely popular for embedded applications, the Cortex-M3 core. This core is based on the ARMv7 architecture, which makes the firmware compatible with many other ARM cores, thus expanding the expected life cycle of the architecture. This performance/cost balanced version of the ARMv7 core is known for its exceptional code density thanks to the Thumb-2 ISA [105], that mixes the 16-bit Thumb ISA with the more powerful ARM 32-bit instruction set. The ARM's closed architecture and backwards compatibility practices reflect on the fact that the Cortex-M3 binaries run on the more recent and powerfull ARM cores like the the Cortex-M4, the recent Cortex-M7 and the high-performance Dual-core Cortex-A8 up to the Cortex-A17, which are all ARMv7 based with Thumb-2 support cores. The debug support, critical for the success of the DBTor implementation, is also ensured with the Core-Sight debug architecture [106] integrated in the core.

The AVR micro-controllers family, developed by the Norwegian Institute of Technology, is a 8-bit architecture, despite appearing in 1996 as one of the first flash-based program memory processor. It is produced by Atmel, acquired by Microchip in 2016 and directed to the small applications systems, with excellent code density characteristics and a quick development tool support [107]. It became very popular because of the amateur electronics platform Arduino, which originally integrated an ATmega8 AVR microcontroller. It consists of a RISC ISA, with a modified

Harvard architecture, allowing program memory accesses from the data address space, however with severe restrictions, as the accesses must use the two dedicated instructions (Load Program Memory and Store Program Memory). Furthermore, the accesses must remain inside the same memory page and within the bootloader section of the code. It also comes with the Atmel integrated proprietary complete debug support based on the JTAG standard. There is also an AVR 32-bit variant, developed to compete with ARM cores, however and despite the superior code density and performance, because it is not compatible with the standard 8-bit AVR ISA neither with the ARM, it never stepped up to the mainstream.

MIPS, originally Microprocessor without Interlocked Pipeline Stages, is a 32-bit ISA for embedded systems [108]. There is also a 64-bit variant of the ISA. The MIPS ISA is used for educational purposes because of its simplicity and consistency. Nonetheless its implementations are quite powerful, which rises its application range to more performance demanding embedded consumables, like telecommunication systems, video-games consoles and digital tv equipments. Its performance is also scalable up to workstations and supercomputers, based on multi-core and superscalar deployments. MIPS cores also allow self-modifying code and code execution from data space memory, which is a *sine qua non* requirement for DBT execution.

The ARMv7-M architecture (Cortex-M3 core) was selected due to its superior performance when compared to the other contenders, regarding the availability of support tools, architectural compatibility and lifetime expectancy. Despite AVR 8-bit architecture being extremely popular and its continuity assured because of the Arduino open-source platform, the fact that it is still an 8-bit architecture is a major disadvantage. Another reason for avoiding the AVR was its memory architecture, which would seriously hurdle the code generation of the DBTor. Although the translator could benefit from the simple and regular MIPS ISA when converting the IR to the target machine code, the fact that ARM has a bigger market-share than MIPS, along with ARMv7 architecture's wide availability influenced its choice over MIPS.

2.2 MCS-51 Architecture

The 8051 [101] is a very popular processor implementing the mature MCS-51 architecture. The interest on this ISA remains today [109, 110], with increasingly

legacy support efforts being developed, either for product lifetime extension or IoT integration. It is an 8-bit pure Harvard architecture (separated program and data memories) with a variable length CISC ISA. It has an internal data space of 256 Bytes, supporting up to 64 KB of external memory (addressable by indirect addressing through a 16-bit (Data Pointer (DPTR)) and a program address space of 16-bit (64 KB). The architecture supports interrupts, timers/counters, GPIO ports, serial communication protocols (e.g.: universal synchronous asynchronous receiver-transmitter (USART)) and other peripherals, however due to the peripherals being tightly coupled in the architecture and mapped in the data memory, many compatibility problems between divergent versions arise. The data memory of the MCS-51 is organized as depicted in Figure 2.1. There are four banked sets of 8 registers in the lower memory region, a bit addressable region and a general purpose RAM area on the bottom `0x7F` addresses. The upper 128 bytes of memory are used for Special Function Register (SFR) allocation. The 8052 variant of the processor includes 128 additional general purpose bytes in the upper memory region, shadowed with the SFR area. The SFR are accessed through direct addressing, while indirect addressing is used to access the general purpose RAM.

The MCS-51 architecture defines an Accumulator (ACC) register located at address `0xE0` for arithmetic logic unit (ALU) operations and indirect addressing. Another important SFR is the PSW, located at address `0xD0`, which keeps the value of condition flags used for conditional branching. Since this register is directly affected by hardware, i.e., its content is not solely manipulated through software, it requires detailed attention in DBT. Figure 2.2 represents the PSW and its bits. From the right to left, the Parity (P) bit, a reserved bit, the Overflow flag (OV) bit, two bits that select the active register bank (RS0 and RS1), a general purpose flag, the Auxiliary Carry flag (AC) bit and the general Carry flag (CY). The parity bit indicates the odd parity of the current value in ACC, the OV bit assigns the existence of overflow in ALU operation, while the CY is used to contain the bit that carries in and out the ACC in some operations, as well as a borrow bit for the subtractions. The AC is used to flag bit carries between the two nibbles of the ACC. These flags' updates must be emulated by software according with the execution status that affect them, and not by direct manipulation from the source instructions.

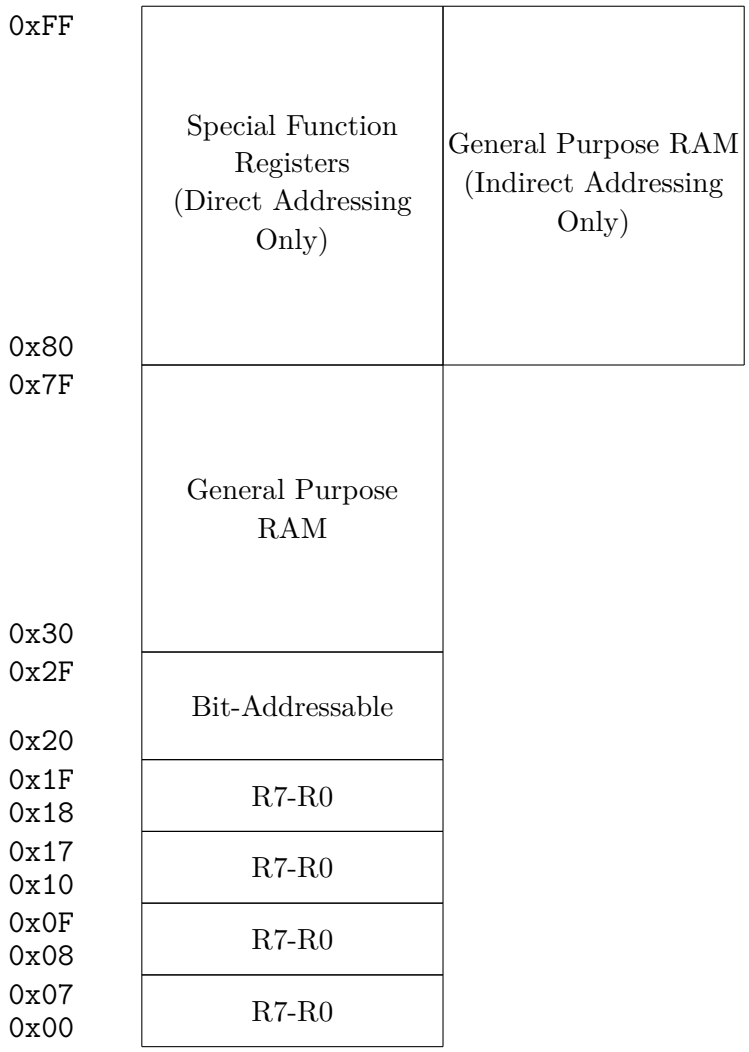


Figure 2.1: 8051 data memory mapping.

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0	
CY	AC	F0	RS1	RS0	OV	-	P	0xD0

Figure 2.2: MCS-51 PSW.

2.3 ARMv7-M Architecture

The Cortex-M3 core, implementing the ARMv7-M architecture, is nowadays one of the most popular microprocessors, known for its excellent C/C++ targeting, highly deterministic operation and, debug and software profiling support [111]. Its broad adoption by the embedded systems arena made its prices drop to very competitive values, becoming a must for modern cost/performance balanced embedded products, such as IoT end-points devices [109]. It follows a RISC design (however with some intricate instructions), on a modified Harvard architecture, sharing the

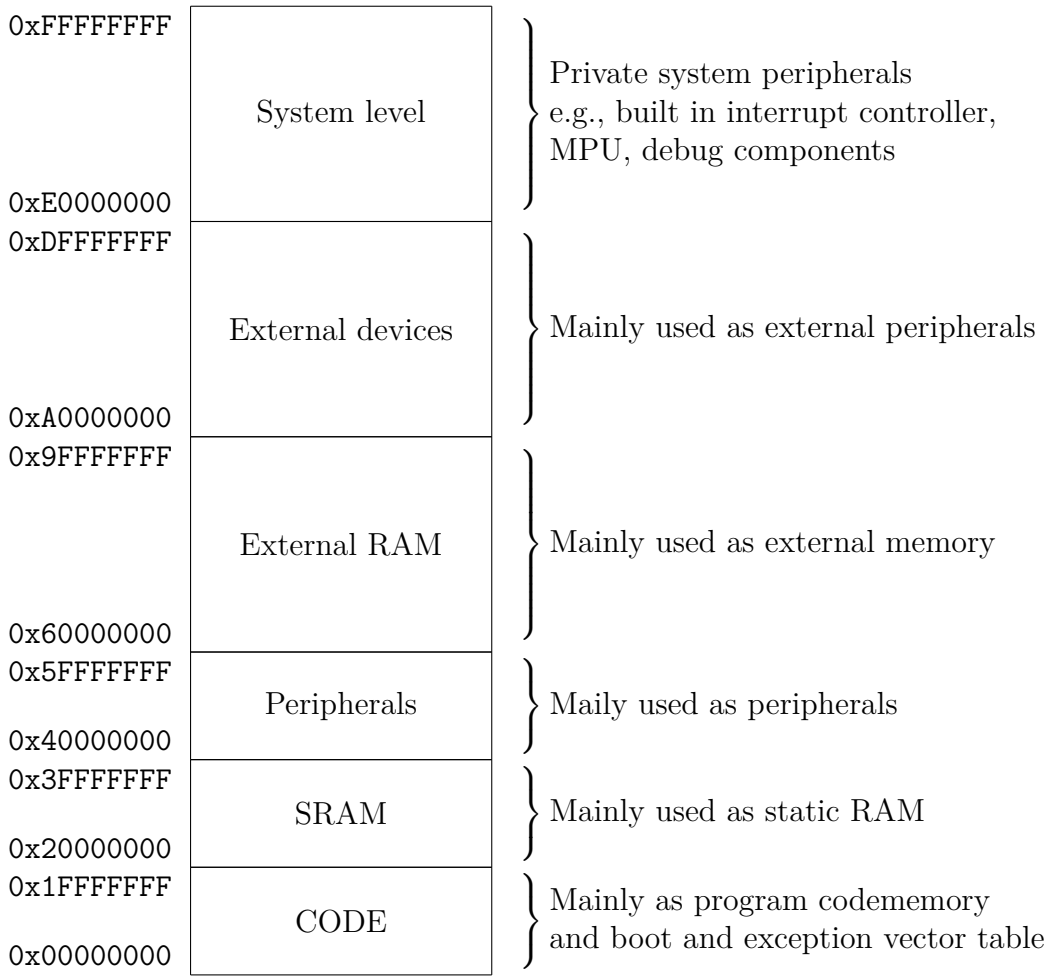


Figure 2.3: ARMv7-M address space.

32-bit address space between code and data buses (Figure 2.3). It supports the Thumb-2 ISA, which allows to mix the compact 16-bit versions of the ARM ISA instructions (Thumb) with a subset of the 32-bit ARM instructions. The ARMv7 architecture conserves the Thumb mode activation bit (present in the ARM ISA), set by addressing the code memory at pair addresses (least significant bit (LSB) is `0b1`).

The program code can be fetched and executed from different memory locations than the "CODE" segment identified in Figure 2.3. Thanks to code and data both sharing the same 32-bit address space, the architecture is capable of self-modifying code execution. It also allows the remapping of some memory sections into code space, to execute code directly from static random-access memory (SRAM) and double data rate memory (DDR) locations without intervention of a loader.

The architecture has a bank of 16 registers, from which thirteen (`R0-R12`) are gen-

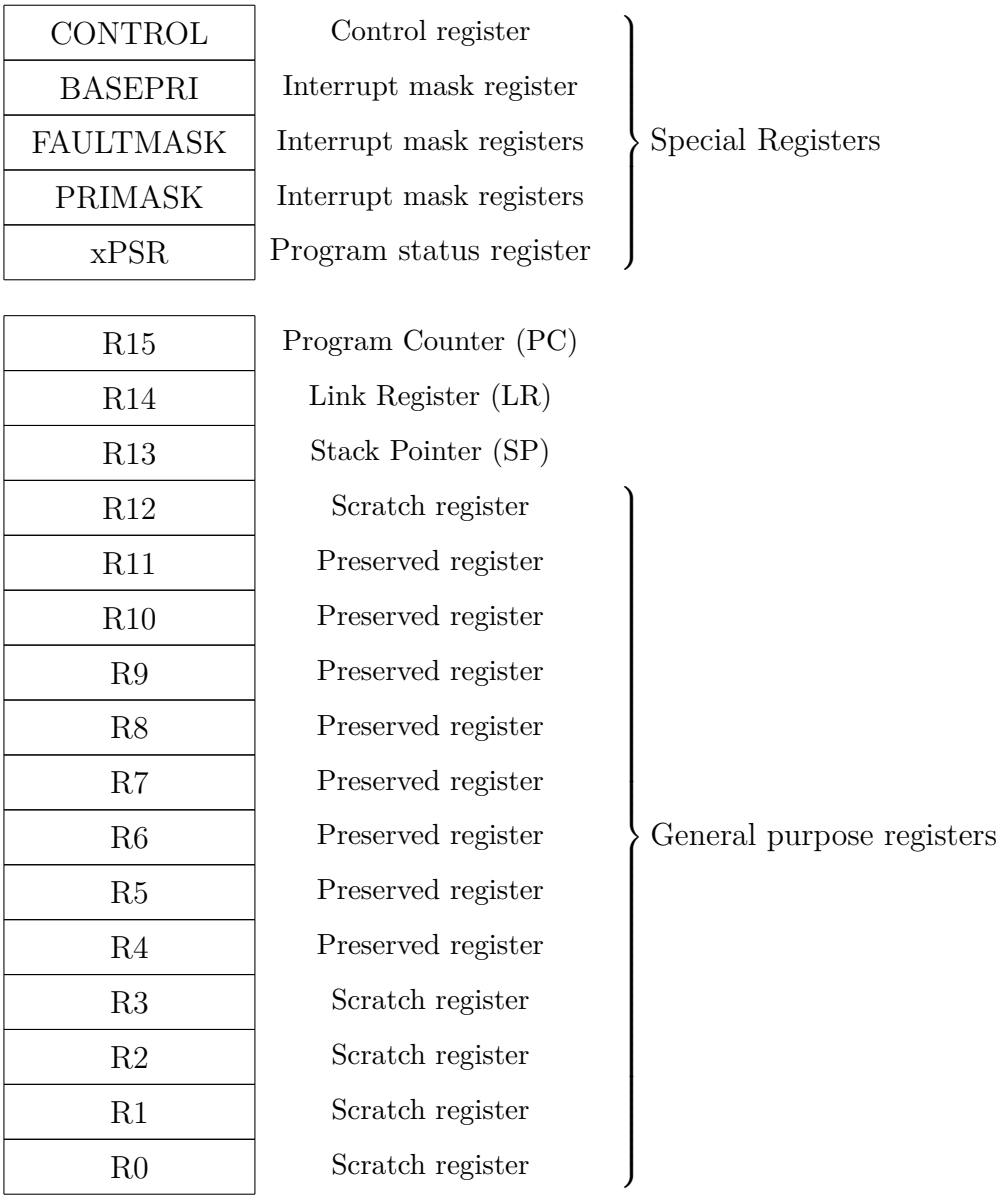


Figure 2.4: ARMv7-M registers.

eral purpose registers, a shadowed Stack Pointer (SP) (`R13`, Main Stack Pointer - MSP, and Process Stack Pointer - PSP), a Link Register (LR) (`R14`) and the Program Counter (PC) (`R15`). The Procedure Call Standard for the ARM architecture [112] separates the thirteen general purpose registers (`R0-R12`) into two sets, according to its calling convention role: Scratch and Preserved registers. Scratch registers' content may be destroyed by any function. Thus, these registers are used to pass and return parameters between *callers* and *callees*. On the other hand, Preserved registers' content must remain the same across functions. Any function that requires any of these registers must save them on entry and restore them prior to return. There are five additional special registers outside

Figure 2.5: ARMv7-M APSR bits.

the register bank: The Program Status Registers (xPSR); three Interrupt related registers, PRIMASK, FAULTMASK and BASEPRI; and the CONTROL register, for privilege mode and stack selection. Figure 2.4 presents the registers in the Cortex-M3.

The Program Status Registers (xPSR) is subdivided in three status registers, the Application Status Register (APSR), the Interrupt Program Status Register (IPSR) and the Execution Program Status Register (EPSR), each respectively used for registering and evaluating conditional flags, registering exceptions in Handler Mode, and conditional execution specific information storage. The APSR, Figure 2.5, has special prominence because of the condition code flags contained and its role in conditional branching. From the most significant bit (MSB) to the LSB there is the Negative flag (N), set when an operation's result in two's complement is negative; the Zero flag (Z) set when a operation's result is zero or a comparison os equal values; the Carry flag (C), set upon carry conditions; the Overflow flag (V), to assign overflow conditions; and the Q bit, used to flag saturation situations of the `SSAT` and `USAT` instructions.

2.3.1 ARM CoreSight Architecture

As aforementioned, the Cortex-M3 core incorporate powerful debug features provided by the ARM proprietary CoreSight Architecture [106]. The CoreSight Debug and Trace components present on the Cortex-M core, depicted in the Figure 2.6, are used together with software debug tools to provide real-time debug visibility to developers and designers. The main components are the Flash Patch Breakpoint unit (FPB), the Data Watchpoint and Trace unit (DWT), the Instrumentation Trace Macrocell (ITM), the Embedded Trace Macrocell (ETM) and the Trace Port Interface Unit (TPIU) [106]. Each of these components play specific roles in the debug of the software and they are briefly explained as follows:

- The FPB implements hardware breakpoints, and provides advanced support for: breakpoints on code addresses and remapping of instructions or literals locations from code or system memory to SRAM addresses.

- The DWT provides watchpoint support for analyzing and monitoring data and code. It is composed by several comparators that can be programmed to match memory addresses and/or their content, PC value and number of clock cycles spent. As a result of the match events it can generate IO signals, trigger debug states or exception, or forward the event to the ITM to be combined with other debug information.

- The ITM is a trace source unit used to provide `printf()` style debug support. The ITM logs debug and trace packages from multiple sources, namely software traces and DWT events, and interfaces them to the TPIU, with optional added timestamp.

- The ETM is a debug component that enables the reconstruction of the software flow, by tracing the instructions that are executed. Software tracing may be started by a DWT event, external inputs or by the Start/Stop control block.

- The TPIU is a debug component that interfaces all the previously described debug modules and an external data stream output. The TPIU associates IDs to the data packets received, synchronizes the packets accordingly with their timestamp, and serializes the information to the debug interface (e.g., JTAG, Serial Wire Debug, etc).

The Cortex-M3 processors integrating the CoreSight architecture support two de-

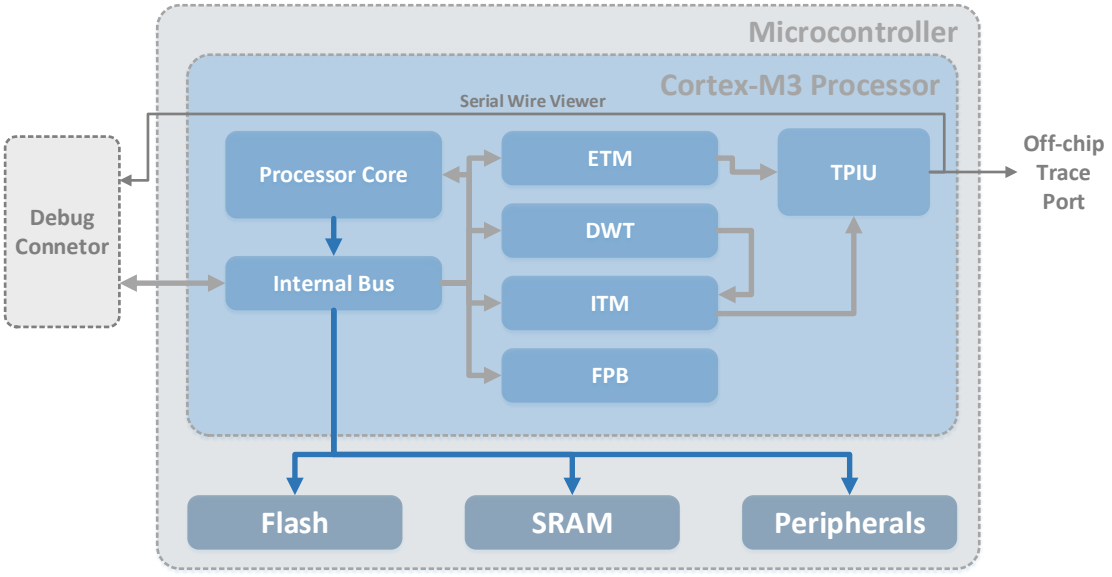


Figure 2.6: Cortex-M3 CoreSight debug architecture.

bug modes: (1) a halting debug mode where core execution is halted for probing and (2) the debug monitor mode, where a debug exception triggers a handler routine to perform the necessary debug operations.

2.4 Development Platform

In order to address the research questions proposed in Chapter 1, deploying a DBTor for COTS low-end embedded system foreseeing hardware support without architectural intrusiveness, and towards the target platform selected in this chapter, a platform that integrates a Cortex-M3 processor is required. The exploration of custom hardware extensions claims for platforms including FPGA fabric. Along these lines, the required development board fall in the category of a SoC, requiring a hard-core ARMv7-M core (Cortex-M3) plus FPGA fabric.

After probing the available platforms on the market, the Microsemi SmartFusion2 SoC was selected. The SmartFusion2 is the Microsemi's next-generation SoC FPGA, with a hard-core Cortex-M3 processor and panoply of peripherals, reliable flash-based FPGA fabric and Advanced Microcontroller Bus Architecture (AMBA) compliant . Microsemi advertises it as the "low power industry leader" (10× lower static power, half of the total power consumption) and preach the SoC's reliability features as the definitive solution for application with high level of criticality (e.g., defense, aviation, medical) [113].

The microcontroller subsystem (MSS) of the SoC includes the Cortex-M3 processor, running up to 166 MHz and supporting low-power modes; 8 KB instruction cache; up to 5 MB embedded SRAM and 512 MB on-chip embedded Non-Volatile Memory (eNVM); a panoply of peripherals, e.g., Serial Peripheral Interface (SPI), universal asynchronous receiver-transmitter (UART), I^2C, DDR Bridges, timers, etc; 5G transceivers with PCI Express endpoints; and dedicated Fabric Interface Controller (FIC) modules to exchange control and data with the FPGA side of the SoC [113, 114].

The FPGA fabric has up to 150K logic elements (LE) of four input, up to 240 digital signal processing (DSP) blocks and configurable large and small blocks of SRAM (up to 240 KB and 4 MB, respectively). To achieve the required reliability for safety critical and mission critical systems, the SmartFusion2 SoC offers state-of-the-art design and data security features: physically uncloneable

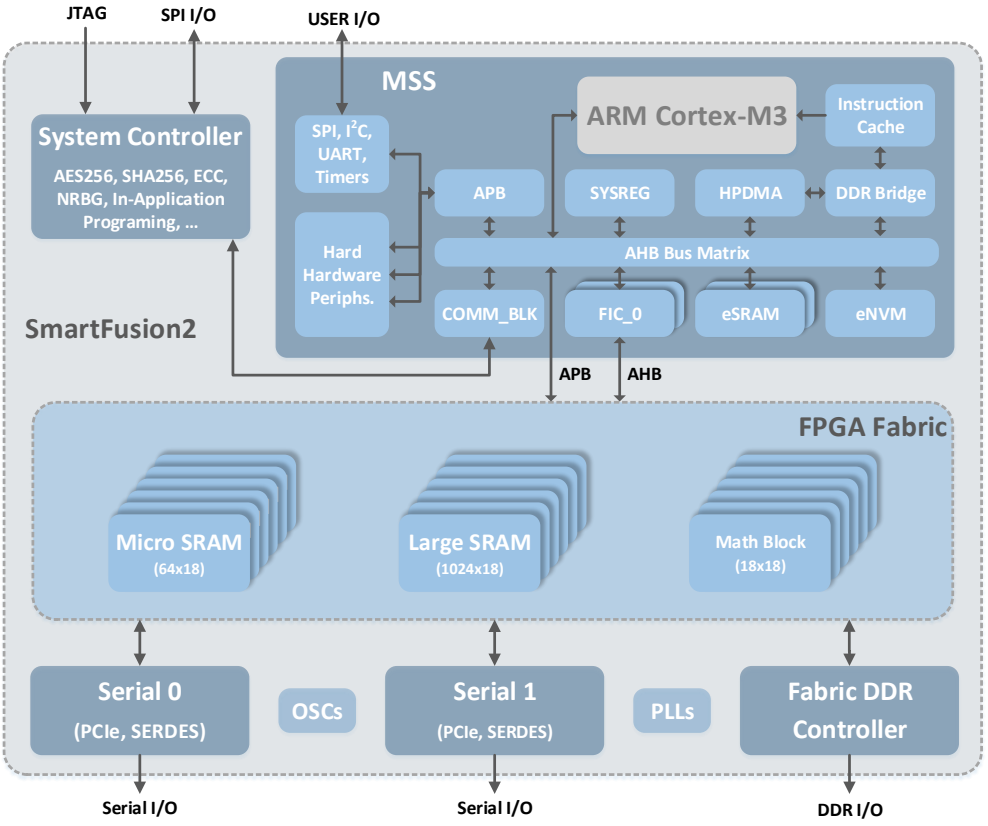


Figure 2.7: SmartFusion2 SoC FPGA simplified block diagram.

function (against tampering, cloning, overbuilding, and reverse engineering), criptographic accelerators for Advanced Encryption Standard (AES), Secure Hash Algorithm (SHA), elliptic curve cryptographic (ECC) and non-deterministic random bit generator (NRBG); single event upset (SEU) immune FPGA technology SEU protected memories (eSRAM, DDR bridges, instruction cache, and peripherals' FIFOs) and SECDEC protected DDR controllers. Once again, despite low power capabilities not being considered in this work, the platform energy characteristics are valued for possible future energy studies. Figure 2.7 depicts a simplified block diagram of the described SoC architecture.

Microsemi provides several development boards and kits with their SoC and FPGA devices. The SmartFusion2 is included in several development kits, each dedicated to different purposes, e.g., from motor control, security evaluation or general purpose development. From the two general purpose development kits available, it was decided to select the SmartFusion2 Advanced Development Kit [1], illustrated in Figure 2.8. The development board features a 150K LE SmartFusion2 SoC FPGA and includes a myriad of peripherals and resources: "PCIex4 edge connector, (...) USB, Philips Inter-Integrated Circuit (I2C), two gigabit Ethernet ports, SPI, and

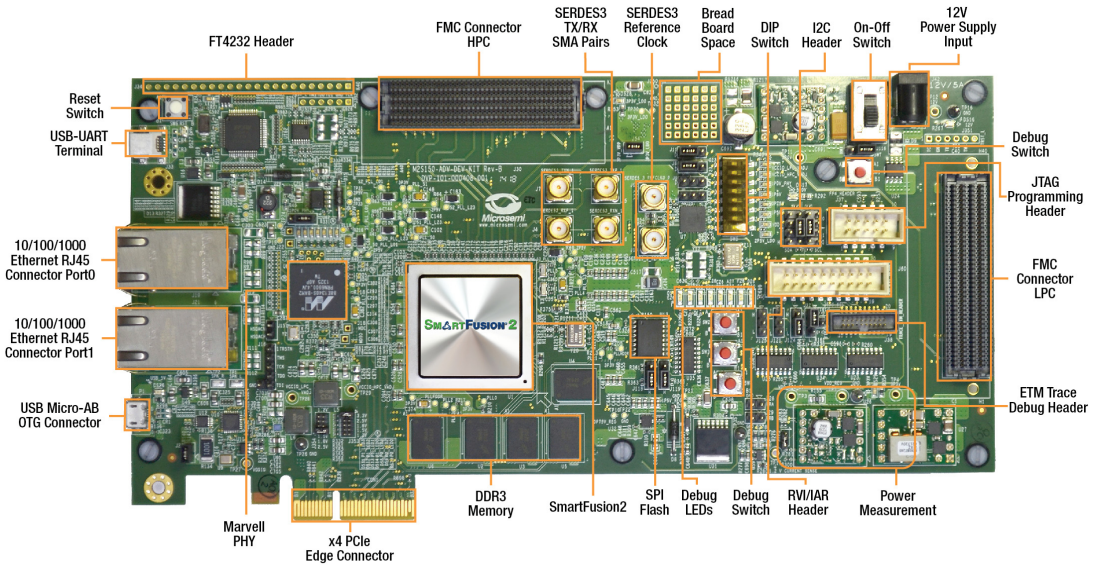


Figure 2.8: SmartFusion2 SoC FPGA Advanced Development Kit [1].

UART. A high precision operational amplifier circuitry on the board helps to measure core power consumption by the device. (...) 1 GB of on-board DDR3 memory and 2 GB SPI flash - 1 GB connected to the MSS and 1 GB connected to the FPGA fabric. (...) Serializer and Deserializer (SERDES) blocks (...)" [1]. The board resources (FPGA fabric and external peripherals) might exceed the requirement of this project and the typical type of systems targeted, however the choice was based on the included device's LE density (150K LEs), driving the concerns away from the board resources sufficiency. Furthermore, the board is dedicated for evaluation purposes and does not impose any obligation in the target platforms.

Microsemi's platforms benefit from the proprietary Libero SoC FPGA design and development software tool. Libero SoC provides start-to-finish design flow guidance and support, integrating electronic design automation (EDA) powerhouses as Synplify, ModelSim, and ViewDraw. It also manages the integration of the system's firmware with common software development environment, including SoftConsole (Microsemi's Eclipse based proprietary integrated development environment (IDE)), Keil, and IAR Embedded Workbench [115].

2.5 Benchmarking Tools

To evaluate how the proposed contributions perform, it is required to provide binaries to the DBTor as inputs. The binaries should replicate the type of programs'

behavior that run in the source architectures. Since this is also the purpose of any benchmark suit - to mimic a typical workload of a system - it was decided to use a benchmark's binaries as the source binaries for the DBTor.

2.5.1 BEEBS

Bristol Energy Efficiency Benchmark Suite (BEEBS) is a set of ten benchmarks ported from other suites, selected accordingly with its type of operations (Branching, Memory intensity, Integer and Floating Point Operations), its applicability for resource-constrained embedded systems and required porting effort, in order to evaluate the energy consumption characteristics of the target platform [116]. It was proposed by Pallister *et. al* at the university of Bristol, United Kingdom and is freely available. Details on the benchmarking applications can be found in the documentation [116]. This benchmark suite, was assembled with benchmarks from the MiBench, DSPstone, WCET, Livermore Fortan Kernels, Dhrystone and MediaBench suites, accordingly to their characteristics, suitability to bare-metal deployment and type of operation variety (Branching, Memory, Integer operations, Floating Point operations). Despite created to evaluate energy consumption on embedded processors, the suite was tailored with the same characteristics of the selected source legacy system: bare-metal implementation, no filesystem, exclude peripherals and small memory footprint. Thus this benchmarking suite was used as a sample set of source application binaries. The source code of the benchmark was compiled to the source architecture, the MCS-51, and used as the source binaries, in order to validate the DBTor's correctness and test the proposed contributions.

The benchmarks were compiled using the IAR C/C++ compiler [117], with the optimizations set to *Medium* for executable size purposes. The optimizations performed in *Medium* are:

- Variables live-dead analysis and optimization
- Dead code elimination
- Redundant label elimination
- Redundant branch elimination
- Code hoisting
- Register content analysis and optimization

Table 2.1: BEEBS benchmarks specs, and compile and simulation results.

Benchmark	Branching	Memory	Integer	Floating Point	Raw binaries size (byte)	Machine cycles
FDCT	H	H	L	H	4443	241175
2D FIR	H	M	L	H	2505	222187
CRC32	M	L	H	L	1791	2355240
Float Maltmul	M	H	M	M	3438	2181894
Cubit Root Solver	L	M	H	L	10061	2487319
Integer Matmul	M	M	H	L	1745	6521313
Dijkstra	M	L	H	L	2272	11320633
Blowfish	L	M	H	L	10018	95973569
Rjindael	H	L	M	L	62340	178125629
SHA	H	M	M	L	4034	528616190

- Common subexpression elimination

However, the optimizations performed in the compiling of the 8051's benchmarks source code have little or no impact (or at least unknown) on the process intend to be marked, because the optimizations impact the generation and execution flow of the source binaries, not the DBTor (thus only on its input), and in a way that it is not perceivable if advantageous or disadvantageous. The benchmarks were also run under the IAR Embedded Workbench simulator, to verify its execution and collect a baseline execution cycles. Table 2.1 shows the type of operation of the benchmark (classified as High (H), Medium (M) or Low (L)), the compiler output and the number of clock cycles obtained from the simulator.

Chapter 3

Designing a Dynamic Binary Translator

The development of any tool is a complex task that requires knowledge, time and resources. Whenever possible existing tools should be used, but when they do not comply with the project requirements, then an implementation from scratch is required. This chapter presents the general architecture of a DBT engine, identifying the functionalities of each component and exposing how they relate and interact in the architecture. Then the design requirements are highlighted and the implementation characteristics that favor such requirements are enumerated, prior to present the design decision taken during implementation. The rest of the chapter is organized as follows: the state of the art is revisited in section 3.1; the generic architecture of a DBTor is introduced in section 3.2 and the design requirements in section 3.3; The deployment insights and the chapter's conclusions in sections 3.4 and 3.5 respectively.

3.1 Introduction

From the literature presented in Chapter 1, many DBT engines may be examined in order to extract the key elements of the architecture and important deployment ideas to successful implement one. In [84], Probst implements a DBTor with an underlaying OS support for application-level emulation. This type of emulation requires system calls conversion from source to the target OS. This procedure is not necessary on a system-level emulation. The author describes the overall ar-

chitecture of the translator, referring to the basic unit of translation which is a *"sequence of instruction likely to be executed as a whole. It has one entry point, and one or more exit point"*. This translation unit for bibliography coherence purposes [44, 45, 69, 74, 88], will be identified from now on as a Basic Block (BB). He addresses the handling of Operands from architecture to architecture, and architecture details that must be handled upon translation, e.g., endianess, immediate size, address space, page size and register mapping. He also introduces the handling of self modifiable and dynamic loaded code. Despite the code being available under GNU general-purpose language (GPL), it targets high-performance architectures such as Alpha, PowerPC and i386.

BOA [42] also describes the architecture of a DBTor for desktop application. Altman *et al.* pretend to optimize PowerPC applications through continuous code profiling to explore ILP execution on a VLIW architecture. This work targets multiple Gigahertz execution and single-source/single-target translation, thus not suitable for using in embedded systems. Moreover, code optimization and tracing strategies used are also very heavy to run on such resource-constrained devices.

In [67], Ling *et al.* describe their approach to DBT on remote thin clients. They analyze the normal DBTor workflow and point the translation misses as a huge penalty inducting factor, proposing to use a client/server distributed architecture to split the functionalities of the translator and expand the code caching capabilities. This approach requires a support network layer, and the thin client, despite the "thin" is a Desktop computer based on a Inter Celeron processor.

Bellard [86] presented the internals of QEMU, a known machine emulator that uses an original and portable binary translation to accelerate the emulation process. In this paper Bellard details how the portable translator was conceived to use an Intermediary Representation, taking advantage of cross-compiler tricks to achieve a portable and retargetable three-address micro operations IR.

The work of [118] describes the optimizations that can and should be performed on a resourceable and retargetable DBTor. The authors identify several overhead sources inherent to DBT (translation overhead, inefficient generated code, code loading, etc) and propose several optimizations such as BB linking, CC handling optimizations and techniques to benefit from multi core execution.

The aforementioned contributions present the architectural requirements and concerns upon the development of a DBT. The authors have also identified sev-

eral sources of overhead and pointed the benefits of a DBTor with resourceable and retargetable characteristics. However, the deployment platforms are high-performance devices rather than resource-constrained embedded systems. Legacy support is mentioned in some of the works, but never as a main application of DBT. This lead to an in-house implementation of a DBTor in accordance with the presented requirements and specifications to explore the possibilities and challenges of binary translation for the resource-constrained embedded arena. This contributes to the state of the art with (1) a DBT engine for embedded systems (2) for legacy support purposes, (3) with resourceability and retargetability in mind, (4) developed following OO paradigm.

3.2 Generic DBT Architecture

The conceptual model of a Dynamic Binary translator is depicted in Figure 3.1. It is generally composed by (1) a DBT engine which contains the Translation and the Execution components; (2) the source binary code; (3) a translation cache/buffer; (4) the (emulated) guest data source; and (5) the target host hardware. Regarding the Execution and Translation, the first refers to the native execution of the translated code, while the Translation can be further split into (6) the Decoder and (7) the Generator sub-components.

Each components' requirements and functionalities can be detailed as follows:

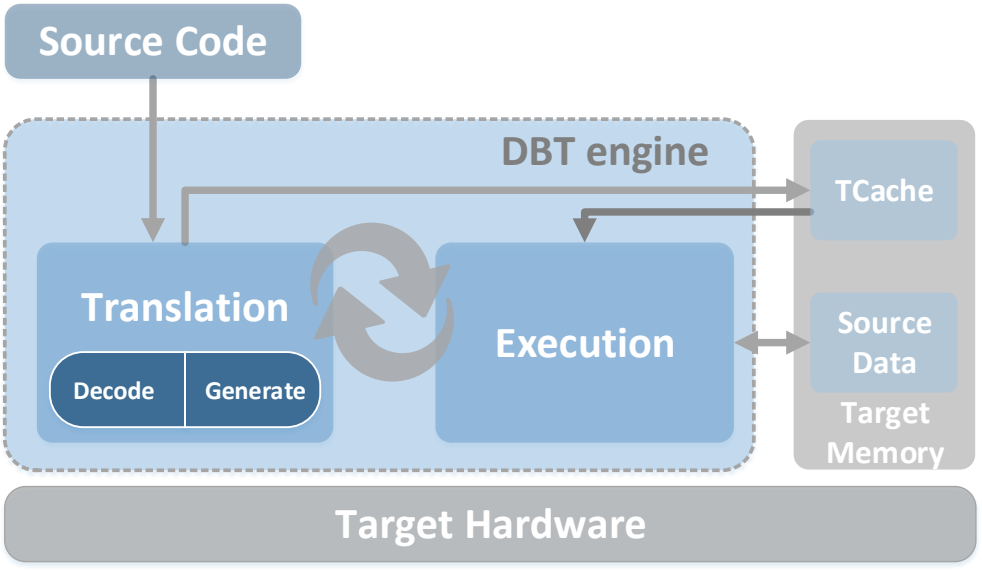


Figure 3.1: DBT engine architectural model.

- **DBT engine** is the representation of the kernel responsible for the DBT operations. It can be understood as the "heart" of the DBTor since it connects and coordinates all the modules involved in the translation and execution of the guest programs. This imperative component must be aware of the available hosting platform resources where it will be running. It hosts the next two main components in analysis, the Translation and the Execution blocks.

- **Translation** is the module that handles the transformation of the source binaries into target machine executable code. This component receives source binaries as inputs, and outputs to a "translation cache", which will be introduced afterwards in this listing. Broadly speaking, the translator's execution is spent between the Translation and the Execution, so this is one of its core components. The code translation is achieved by reading the source binaries, one instruction at a time, decoding its meaning and generating target machine code to reproduce its behavior. These transformations can be achieved through different ways, however in order to decouple the source and target implementation-specific parts, two other sub-components are needed: the Decoder and Generator. Since the source/target architecture bridging is handled in this component, it comprises most of machine specific support code, i.e., the source ISA decoding and the target machine code generation, respectively.

- **Decoder** is one of the Translation sub-components. This component is purely dedicated to interface the source binaries, thus it is source specific. It must distinguish each of the source ISA instructions and decode their meaning in order to transmit it to its "pair" component, the Generator. The decoding of the instructions can be implemented through different techniques and algorithms, but in order to not affect non-source related components, there must be some sort of common interface between this component and the Generator. In doing so,, modifications on the source-specific decoding have no impact on the target specific code generation, independently of the implementation. It is common to use an IR between host and guest ISAs to perform such interface and isolation.

- **Generator** is the mentioned pair of the Decoder component, dedicated to generate target machine code. Contrary to a code simulator, the generator must output executable code to the target machine based on parameters

transmitted by the Decoder. The transfered information contains data relative to the operation type, operands, immediate values, branch destination, etc. The generated code must replicate the exact behavior of the source code, otherwise the execution flow will be lost. This is the Translator's target ISA specific component, thus must be in accordance with the host hardware in use. It is the major part of the DBTor to be modified in case of porting to other target architecture.

- **Tcache** or translation buffer, together with the Generator, is one of the components that mainly differentiates a translator from a simulator, and it is where the translated code is stored before being executed. This component must be allocated on a section of the host machine's memory with write access and execution privileges. It must implement content management and replacement mechanisms to accommodate the incoming translations from the Generator and keep track of the stored BBs.

- **Execution** is not a tangible component (i.e., with implementation), but rather the native execution of the generated code. However, and as introduced in the "Translation" description, the DBT engine is continuously switching between code translation and code execution, thus this component must be part of the architectural mode. The code is executed one TBB at a time and after completion, the Execution gives its way to the Translation, successively.

- **Source Binary Code** represents the binaries that were generated to be originally executed in the source processor architecture. This is the code intended to run on another architecture than the one it was compiled or assembled for. The code format can be raw binary, `.hex`, or any executable file format such as `.elf` or `.exe`, depending on supported formats. The provenience of the code may also be diversified, e.g., from a non-volatile memory, received through a communication interface, or gathered from the network. These are all applications specific details, however due to code translation requiring quick and repetitive accesses to the code, some sort of memory hierarchy structure, like a cache or a buffer is required to reduce access latencies.

- **Source Data** or Guest Data is the guest related memory emulation support. It contains items such as the guest Program Counter and other special registers, execution information and the emulated guest memory. This memory

must be accessible either during Translation or during Execution stages.

- **Host Hardware** or Target machine is the processor where the DBTor runs, that is to say, the architecture for what the DBTor must be compiled for, as well as the target machine for the code obtained from the Generator. The affinity between this machine and the Source ISA establishes most of the implementation requirements and specifications.

When considering a novel implementation, the presented elements were classified as "source/target variable" or "DBTor intrinsic. The elements

(a) DBT Engine;

(b) Execution;

(c) Source Binary;

(d) Translation Cache;

fall into the "DBTor intrinsic" category because their architectural design requirements do not depend on the guest or host architectures. Thus their implementation and integration should be the same among different Source ISAs and Target Hardwares.

The remaining elements, i.e.,

(a) Decoder;

(b) Generator;

(c) Source Data support;

are ISA specific, thus considered "source/target variable" and require re-writing effort to comply with different ISAs. Generally, every implementation should consider the target's available platform resources.

3.3 Requirements and Design Decisions

The implementation requirements that must be present upon architecture design and implementation of the DBTor were gathered within the scope of this work. Such requirements comprise:

1. DBT targeting resource-constrained embedded systems. The system deployment must be kept minimalistic for a low memory footprint, avoiding heavy processing algorithms, complex algorithms and making the most common case fast.

2. A system level DBTor for full system emulation. In these types of translators, the DBTor does not provide any system support, contrary to the application level translators. This implies that if the source binaries where compiled for an OS, then the full system image must be loaded and translated.

3. There are no dedicated reasons that justify the use of an OS (i.e., parallelizeable tasks, concurrent accesses to computing resources or peripherals, network stack or user interface). Thus, the DBTor should be a bare-metal solution, sparing unnecessary resources use by the OS.

4. The DBTor should provide data support structures for the source binaries and translation fragments.

5. Resourceability and retargetability are two major requirements. Although the variability associated with these concepts might penalize the performance, considering the application range and the type of source binaries involved, some performance loss is acceptable. The benefits of an architecture designed with those in mind are ease of modification, severe code reuse and enforced correctness by design upon supporting new ISAs.

6. High level abstraction without neglecting code efficiency, inherited from the previous requirement.

3.4 Deployment Insight

With the exposed requirements in mind, C++ was the elected programing language to implement the system. It is a proven OO programing language with efficient machine-code generation, and inherent variability management and code reuse mechanism like **Inheritance** and **Polymorphism**. The penalty induced by the dynamic characteristics of the language might be contained according to Cardoso [119] and static virtual table analysis [120].

The algorithm was implemented from scratch with re-usability in mind. The variability points where identified and managed through C++ features and techniques,

in order to ease DBTor's configuration and multi source/target translations support. The following descriptions will detail how the module's variability was handled during implementation. Despite presenting implementation details, the purpose of this section is towards variability management techniques used and not as an implementation guide of the project.

3.4.1 Retargetability Support

In order to achieve easy retargetability, the source and target specific implementation must be decouple from each other. This approach has three implications:

1. The DBTor's kernel functionalities must be apart from the source/target's implementations in order to avoid dependencies among the DBTor's generic functionalities and modifications performed to the source/target ISAs implementations;

2. Source and target functionalities and implementations must be independent from each other, avoiding modification to both source and target implementations in case of a new source or target ISA support.

3. An IR interface between the source and target ISAs for information exchange between the Source instructions' Decoder and the Target's code Generator.

Intermediate Representation

A QUEMU-based IR [86] was used for its simplicity. The IR is a defined set or virtual instructions which must be output by the Source architecture Decoder, and must be ported to the Target architecture generator. Each micro operation is represented by a C++ function that must generate target machine code that reproduces the behavior it represents. This equivalence must be established manually or automatically, by mean of an external tool, but ISA automatic pairing falls outside the scope of this work. The virtual ISA of the IR is based on a 3-address machine, with the following operations (Listing 3.1):

```
//conditional IR instructions
virtual void gen_cmp(uint8_t SReg1, uint8_t SReg2) = 0;
virtual void gen_cje(uint8_t SReg1, uint8_t SReg2, uint8_t offset) = 0;
virtual void gen_cjne(uint8_t SReg1, uint8_t SReg2, uint8_t offset) = 0;

//branch IR instructions
virtual void gen_writePCreg(uint8_t SReg) = 0;

//data transfer IR instructions
virtual void gen_mov(uint8_t DReg, uint8_t SReg) = 0;
virtual void gen_movi8(uint8_t DReg, uint8_t imm) = 0;
virtual void gen_movi16(uint8_t DReg, uint16_t imm) = 0;
virtual void gen_movi32(uint8_t DReg, uint32_t imm) = 0;

//load from emmulated memory IR instructions
virtual void gen_ld8(uint8_t DReg, unsigned int imm) = 0;
virtual void gen_ld16(uint8_t DReg, unsigned int imm) = 0;

//store to emmulated memory IR instructions
virtual void gen_st8(unsigned int imm, uint8_t SReg) = 0;
virtual void gen_st16(unsigned int imm, uint8_t SReg) = 0;

//alu IR instructions
virtual void gen_add(uint8_t DReg, uint8_t SReg1, uint8_t SReg2) = 0;
virtual void gen_sub(uint8_t DReg, uint8_t SReg1, uint8_t SReg2) = 0;

virtual void gen_div(uint8_t DReg, uint8_t SReg1, uint8_t SReg2) = 0;
virtual void gen_mul(uint8_t DReg, uint8_t SReg) = 0;

virtual void gen_not(uint8_t DReg, uint8_t SReg) = 0;
virtual void gen_or(uint8_t DReg, uint8_t SReg) = 0;
virtual void gen_and(uint8_t DReg, uint8_t SReg) = 0;
virtual void gen_xor(uint8_t DReg, uint8_t SReg) = 0;

virtual void gen_shr(uint8_t DReg, uint8_t SReg1, uint8_t SReg2) = 0;
virtual void gen_shl(uint8_t DReg, uint8_t SReg1, uint8_t SReg2) = 0;
```

Listing 3.1: IR micro operations declared as pure virtual methods.

The arguments present in each of the above micro operations functions represent either an immediate value (e.g., `offset`, `imm`) or a temporary working register (e.g., `DReg`, `SReg`). The immediate arguments are literals extracted from the source binaries while the temporary working registers refer to target machine's general purpose registers used to emulate the 3-address machine. This approach was used

instead of direct register mapping to favor portability to target architectures with low general purpose registers count.

The porting of these IR will be introduced in the next chapter.

Source and Target separation

In order to provide the mentioned separation between Source and Target implementations, while promoting code reuse, the C++ concepts of Inheritance and Abstract Base classes were resorted. Defining a base class **Translator** that implements all the "DBTor intrinsic" functionalities as base methods promotes its reuse for every instance. Additionally, the base class also establishes the porting interfaces as pure virtual methods. With this last mechanism is forced the implementation of the IR virtual instructions and the Source and Target specific methods, while blocking the declaration of a Translator object without them. The source-related virtual method is the `decode()` and the target-related are the implementation of each IR micro operations, which represent the **Generator**.

In order to port the DBTor, a derived class of the **Translator** must implement the source and target ISA methods, and then it is ready to be compiled for the target architecture. This method allows flexibility during implementation, while respecting the general guidelines (main class methods). Although this type of dynamic polymorphism is known to cause performance penalties, since there is a single Derived class, the penalty can be contained through "virtual table elimination" compiler options [121, 122].

Below in Listing 3.2 is presented the header file of the **Translator** class, from which the composing elements will be explained ahead in this chapter. The abbreviated IR micro operations were presented in the code snippet before and removed here to avoid repetition.

```
1  #ifndef TRANSLATOR_H
2  #define TRANSLATOR_H
3
4  #include "TransFlushCache.h"
5  #include "CBuffer.h"
6  #include "types.h"
7
8  typedef struct SourceEnvironment {
9      SOURCE_PC PC;
10     uint8_t * dataMem;
```

```
11 } SourceEnvironment;
12 class CTranslator
13 {
14 protected:
15     SOURCE_MEM_BASE * pSourceProgMem;   //source program pointer to the
16                                         //beginning of the loaded binaries
17     CTransFlushCache transCache;        //Translation cache instance
18     CBuffer codeCache;                  //CCache with the source in use
19
20     uint8_t volatile * currBBExecPtr;   //pointer for the BB during decode
21     SourceEnvironment volatile env;     //source enviroment data struct
22
23     //translation support variables
24     bool volatile eoExec;               //End of Execution flag
25     bool volatile eoBB;                 //End of Basic Block flag
26
27     //Pure Virtual methods
28     virtual void envReset(void) = 0;    //implemented by the child class
29     virtual void decode(void) = 0;      //decode and fetch instruction
30                                         //into a set of micro Ops.
31     virtual void gen_prolog(void) = 0;
32     virtual void gen_epilog(void) = 0;
33
34     //Virtual IR interfaces
35     /*
36     ...
37     */
38 public:
39     CTranslator(int Ccache_size, int TcacheSize);   //constructor, with
40                                                     //initialization
41                                                     //parameters
42     ~CTranslator(void);
43
44     int reloadTranslator(void * sourCodeAddr , int sourCodeSize ,int
       exit_address);                                //initialization and reset
45                                                     //method
46     int sourceCodeLoader(void * programStart);  //load source binaries
47                                                     //into codeCache
48
49     int runDBT(void);                   //method that runs the DBT engine
50 };
51
52 #endif
```

Listing 3.2: Translator class header file.

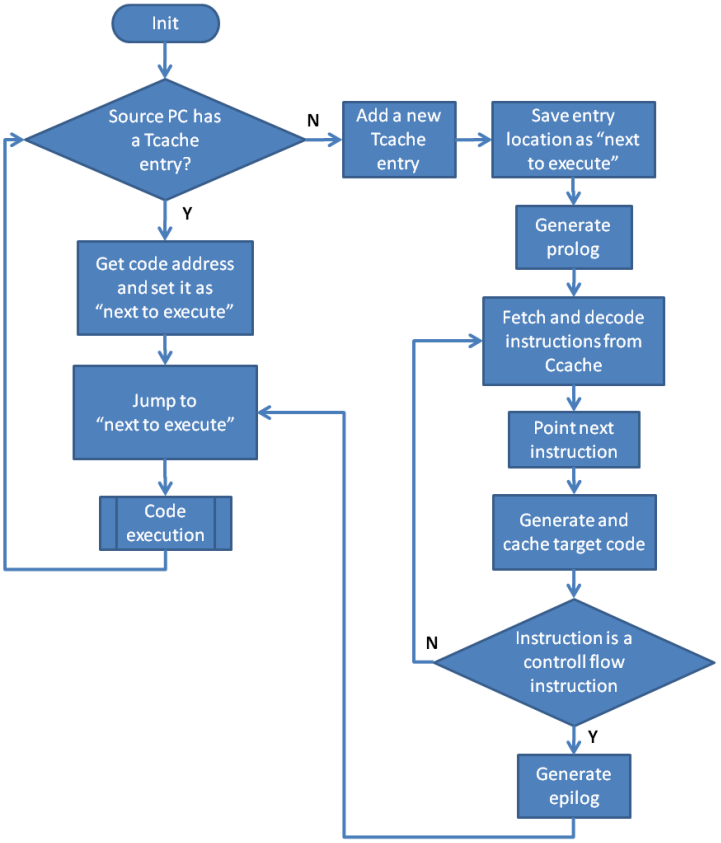


Figure 3.2: DBT engine flowchart.

3.4.2 DBT Engine

The developed DBT engine implements the algorithm illustrated in the Figure 3.2. The DBT engine manages the Translation and Execution processes of the source binaries, which is handled in units of BBs, where each BB is identified by its first instruction's address, i.e., the source architecture's PC value. Each source BB after translation will generate its equivalent translated Basic Block (TBB), saved in the Tcache. The TBBs are saved in the Tcache and in addition to the target code, have a prologue and an epilogue, responsible to save and restore the Translation/Execution context, respectively

The translation process starts with a query of the source PC value to the Tcache. If the BB was already translated and its corresponding TBB is in cache, the Tcache address where the TBB is stored is retrieved and the DBT engine switches to the Execution environment in order to run the TBB block. At the end of the TBB's execution, the epilogue will return the execution to the DBT engine and perform another query to the TCache for the next source BB to translate and/or execute.

The PC value is affected by conditional branches and function calls during the execution of the TBB. If the source PC is not yet translated, then a new translation must be executed. This is started by adding a new entry to the Tcache for the new BB and saving the address (for later execution) where the generated code of the TBB will start. Then the prologue is stored in the Tcache and the Translation process starts its loop until fetching a control flow instruction. Each instruction is decoded and translated individually and the PC updated to the next value, accordingly with the instruction length. A control flow instruction determines the end of the Translation cycle and the generation of the TBB's epilogue. The previously saved TBB location address is now used by the DBT engine to switch to the execution of the recently translated BB.

3.4.3 Translation to Execution Switch

A seamless and portable Translation/Execution context switch is supported by the C/C++ execution environment, handled as a function call and return. When switching from Translation to Execution, the TBB's start address is casted as a function pointer and a call is performed through it. This method ensures that the C compiler will generate the necessary code to jump to the TBB and start its execution. After the function call, the *callee*, i.e., the TBB, must respect the target architecture's application binary interface (ABI) regarding parameter passing, return address and context saving. These tasks are left to the generated prologue at the beginning of the TBB. At the end of the TBB, the generated epilogue will handle the context restore and the return of the values and the execution to the *caller*, which is the Translator's Execution. This is handled like a common function call and return process. The single porting effort remains on the adaptation of the prologue and the epilogue accordingly with the target's C++ ABI.

```
1 ...
2     //SWITCH TO EXECUTION
3     ((void (*)(int*))((int*)(currBBExecPtr+1)))((int *)(env.dataMem));
4
5     //RETURNED TO TRANSLATION
6 } while( !eoExec );
7 ...
```

Listing 3.3: Translation to Execution switch code snippet.

The Listing 3.3 shows a code snippet of the TBB's start address casted as a function, inside the Translation cycle.

3.4.4 Memory Support Structures

There are several memory support structures involved in the process of DBT. The DBTor is a memory intensive system due to the repetitive accesses to the source binaries and the target code storage and access. Memory support structures like caches and buffers are used on a two fold way: (1) to cope with memory latencies associated with accesses and (2) as a storage support entity for data and code. Additionally, it is required a structure to contain translation related source data (e.g., PC, location of emulated source data memory, etc.), called the source environment data. The three different used memory support structures used are detailed in the next paragraphs.

Source Binaries Cache

The Source Binaries Cache, or source code cache (Ccache) purpose is to store the source program on system memory and set it available byte by byte to the DBT engine upon request. On the embedded DBT context this module exists for several reasons, such as:

1. Quick access - The binaries are usually stored in flash memories or other ROM technologies, which typically have associated access latencies. These access costs can be alleviated with the use of a proximity storage buffer on system RAM. The designated memory space should be pre-loaded during DBTor initialization, thus removing the ROM access costs from execution;

2. Size constraints - The source binaries size may not fit on the available system RAM. This condition may be overcome with the use of external memories or other type of storage alternatives. In order to execute binaries larger than the available system memory, the cache is partially loaded and a *hit/miss* condition verification is assessed during the runtime `load()` method to fetch a new block of binaries from its external location, leading to the next reason;

3. Flexibility - The source binaries may be loaded from a variety of external locations such as external Flash memory, the original storage support of

the binaries or transfered though any communication interface (e.g., RS-232, Parallel port, SPI, etc). The single porting effort is focused on the `load()` method, which must implement the interface and the data transfer from the external source to the internal storage buffer.

4. Variability management - Because the DBTor always reads the binaries from the Source Binaries Cache, the variability associated with supporting multiple external binaries' sources is isolated from the DBTor algorithm. In this fashion the variability point is reduced to one (`load() method`), and apart from the DBT engine.

For these reasons the Source Binaries Cache was implemented as an individual class and integrated in the Translator class by **Composition**.

Translation Cache

The Tcache, also called Fragment cache [98], is the storage entity where the TBBs are stored to be executed. Besides the storage functionalities it relates the original BBs with the location of the TBBs. This association between source and target addresses is essential to DBT process because it records the equivalence between source and target PCs, and consequently BB and TBB. Because of its characteristics, this entity requires different features from the Source binaries cache, which are described bellow:

1. Write and Execution Privileges - In order to store and natively execute ISA source code (TBBs stores in memory), two conditions must be fulfilled: (a) the memory where the TCache is allocated must be rewritable during runtime and (2) it must have execution privileges. These two condition are not always simultaneously true, thus, the target candidate must have a memory with simultaneous write and execution privileges. That is, a target machine implementing a pure Harvard architecture (with separate program and data memories) would offer serious obstacles to support DBT.

2. Content Management - The Tcache requires a data structure to manage and locate the stored TBBs. This has two purposes: (1) check if the queried BB was already translated and (2) return the location of the TBB. In case of a *hit* (i.e., the BB was translated and the TBB is stored in the Tcache) the TBB's address must be returned to the DBT engine to be executed, otherwise a *miss* is passed to the DBT engine to start a new translation. There are multiple

suitable data structure for this purpose, and its complexity and overhead must be considered. Implementation apart, the Tcache inputs a source PC (i.e., BB base address), which should be used as a unique key to locate the equivalent TBB.

3. Eviction Mechanism - When the Tcache reaches its maximum capacity, its content must be evicted in order to accept new TBB. This might be a partial or total eviction (i.e., *flush*), depending on the implementation, however partial eviction requires coherency enforcement when directly linking BB during translation, which might contribute with excessive overheads.

In accordance with the approach followed for the Source Binaries Cache, this component was also implemented as a standalone class and then included in the Translator class by **Composition**. In this way different Content Management and Eviction Mechanisms can be attempted without modifications to the DBT engine nor the Source and Target specific components.

Source Environment Support

This last support structure is used to accommodate source runtime relevant values, such as the source PC, the location of the source data memory(ies) location(s), special registers and values that must be accessible during either Translation or Execution. Because of this access availability requirement, and since it is used a C++ standard function call procedure to switch between Translation and Execution, it was implemented as a **struct** in the Translator class. This allows quick direct accesses from both sides when passed to Execution by **reference** a function parameter.

3.4.5 Helper Function Calls

The Helper Functions are a mechanism *à la QEMU* used to emulate source program behaviors which do not have a direct translation to target machine code, or are difficult or impractical to translate. The behavior intended to be emulated is described in a standard C++ functions, which is then compiled and loaded to memory together with the DBTor. Upon execution (in the Execution context), the helper function call and return must respect the C++ calling convention, and register's usage as described in the architecture's ABI. In order to do so,

a Helper Function Call generator `gen_helper(void (CTranslator::*helper_function)(void))` is used to generate and assemble the call instructions with the helper function address. The generated code is stored into the Tcache and during Execution it performs a conventional function call. This strategy is used for tasks such as source calls (handling the stack, the return address, parameters, etc), for debugging and instrumentation purposes and some CISC instructions emulation. The implementation of Helper Functions is left for the target specific porting of the translator, and performed according to deployment necessities.

3.5 Conclusions

In this chapter it is explained the requirements and design decisions taken upon the implementation of a DBTor dedicated to resource-constrained embedded systems. The variability points where identified and its management through C++ features explained. The DBTor description will be completed in the next chapter, where the source and target architectures proposed in chapter 2 will be paired and the resulting source-target use-case tested.

Chapter 4

Case Study: 8051 on Cortex-M3

In order to evaluate the design and attest the operation of the demonstrator is necessary to proceed to the final part of the implementation of the DBTor, the porting of source and target architectures selected in Chapter 2. This will allow to evaluate DBT on the tightly constrained embedded systems' domain and explore the opportunities offered by this technique. The previously established constraints and decisions regarding the code isolation and ease of portability were followed and this chapter describes the implementation process of the source and target parts of the project, guiding future porting efforts of the DBTor. In the remaining of this chapter, Section 4.1 approaches the source and target specific details of the porting effort, together with the conditional flags handling. Section 4.2 refers to the testing and verification of the translator and Section 4.3 presents the conclusions of the chapter.

4.1 Pairing the MCS-51 and the ARMv7-M Architectures

In this section it is demonstrated how the source and target architectures were articulated together, detailing each architecture's porting tasks (i.e., MCS-51 and ARMv7-M).

The DBTor executable is a C++ object that inherits all the base Translator class methods and attributes, along with the implementations of all the source and target related virtual methods. Aside from the virtual methods implementation,

there are additional aspects regarding the porting operations to be considered, namely the target architecture's ABI. The C++ calling convention and registers usage scheme related to the target architecture must be known in order to mimic a native C++ context switch (between Translation and Execution). The target architecture's ABI encompasses the following information: 1) the return address location and return procedure to follow by the *callee*; 2) the parameter passing scheme from a *caller* to the *callee*; and 3) the architecture's register usage during subroutines' execution. The relative importance of each topic is addressed as follows:

1. The first topic concerns the prologue and epilogue of the translator. These two DBTor' features must replicate a standard *callee* entry and exit behavior, so that the Execution of the translator is handled transparently as a regular C++ function. The Procedure Call Standard for the ARM Architecture (AAPCS) [112] designates that subroutine calls use the primitive subroutine call instruction BL (branch and link), which loads the next value of the program counter to the LR and the destination address to the PC. The control is returned to the *callee* by transferring the value of the LR to the PC. The prologue of the TBB must preserve the LR value for the epilogue to return the execution to the Translation.

2. The second topic purpose is to ensure parameter passing between Translation and Execution in accordance with the C++ standard, i.e., parameters location, size and order. The AAPCS' information on parameter passing is extensive and detailed, however the information to retain is:

   (a) the base standard for passing up to four arguments to a subroutine is in core registers R0 to R4. For more than four arguments, the excess is passed through the stack;

   (b) variables up to 4 bytes are designated to an individual 32-bit general purpose register;

   (c) the arguments are passed accordingly with the order they are specified in the function call;

   (d) in C++, an implicit pointer of the base class (`this`) is included when a method is called and precedes the first user's argument.

   The code snippet in the Listing 3.3 from Chapter 3, displays how the Trans-

lation uses a function call to switch to Execution, by casting the TBB start address as a function pointer. The function has one argument (the source architecture data memory pointer), which must be handled in the Execution environment accordingly with the described above. Helper functions calls must also respect the argument passing convention.

3. The last topic guides the choice of registers used in the Execution as temporary or working registers. The ARM architecture distinguishes the general purpose core registers as *preserved* or *scratch* registers (refer to Figure 2.4). Scratch registers, `R0-R3` and `R12`, may be freely used by the subroutines without caring about their previous contents. If any of these registers is required after a function call, the *caller* must save it during the procedure call. On the other hand, preserved registers' value must be preserved across function calls. The *callee* may use these registers but their content must be saved before use and restored at the end. During interrupts and exceptions, when unexpected context switch occur, scratch registers are automatically stacked by the architecture in order to prevent them from being lost during the handler routines. The values are restored upon interrupt/exception return. Thus, the correct use of registers during Execution must have into account the type of register:

   (a) scratch registers (`R0-R3`, `R12`) may be used "on demand" and do not interfere with the Translation's context;

   (b) preserved registers (`R4-R11`) use requires additional prologue and epilogue operations for their save and restore;

   (c) the calling of helper functions from the Execution side requires the preservation and restore of scratch registers, if in use.

4.1.1 Source Specific Porting

In this subsection the necessary source architecture porting tasks are described. The porting was grouped in three groups of tasks, and it will be explained together with code snippets that exemplify the implementation.

Source Instruction Decoding

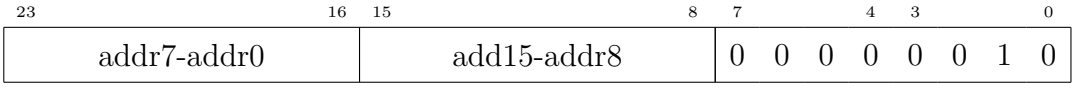
This operation is essential to the translation process, and the first of a long chain of operations that constitute the DBT. In order to be translated, the source ISA instructions must be recognized by the translator. The desired behavior of the source instructions decoding resembles a common disassembler with the difference that its output should be the DBTor' IR, not assembly code. This task is accomplished through the implementation of the `decode()` virtual method in the child class of the Translator class. The `decode()` method fetches bytes from the source binary memory structure and identifies the corresponding opcode, how many bytes compose the instruction (MCS-51 has a variable length ISA), identifies the operands and isolate them. This is subject to different implementations.

Chen *et al.* [13] implemented an efficient code generator algorithm for binary translation, following a graph based approach. Authors claim that using graphs to map source to target instructions, their system is able to translate code with many-to-many instructions mapping, contrary to most of the translators, which follow a one-to-many instructions mapping. The methods used to construct the data flow graph of the binaries to graphs were not disclosed in the publication. This mechanism was implemented on a dual core ARM Cortex-A8 for user level DBT. The presented results encourage its use, but due to the processing power discrepancy between Chen's and this thesis's target systems, the method was not followed.

For simplicity purposes, the decoding was implemented through a nested `switch` expression sequence. The main `switch` evaluates the most significant nibble of the first byte of the instruction, and the derived `switch` statements evaluate the least significant nibble, until the instruction under analysis is identified. The base approach to binary translation should have no hardware assistance and it is focused on software-only solutions that fit in COTS products. The code Listing 4.1 shows parts of the decoding method implementation. The `FETCH` symbol is a macro for reading one byte from the source binaries and incrementing the source binaries pointer (source PC). The method executes until finding an instruction that assigns the (`eoBB`). The snippet displays the nested `switch` structure and the decoding of three instructions: long jump, `LJMP addr16`; return from subroutine, `RET`; and a sum of an immediate with the accumulator `ADD A, #imm8`. The nested switch structure expands until covering all the MCS-51 instructions.

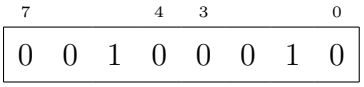
```
1  void CTranslator8051::decode(void) {
2      uint8_t op;
3      uint16_t tmp1, tmp2, tmp3, tmpAddr;
4
5      while( eoBB == false )
6      {
7          op = FETCH;
8          switch(op&0xF0){
9          case 0x00:
10           switch(op){
11                 case 0x00:
12 ...
13                 case 0x02:// LJMP addr16
14                     tmpAddr = FETCH;
15                     tmpAddr <<= 8;
16                     tmpAddr |= FETCH;
17                     gen_writePC(tmpAddr);
18                     eoBB = true;
19                     break;
20 ...
21         case 0x10:
22 ...
23         case 0x20:
24           switch(op){
25 ...
26                 case 0x22   :// RET
27                     gen_helper(&CTranslator8051::helper_ret);
28                     eoBB = true;
29                 break;
30 ...
31                 case 0x24   : // ADD A, #imm8
32                     tmp1 = FETCH;
33                     //Load temporary registers
34                     gen_ld8(tReg1, A);
35                     gen_movi(tReg2, tmp1);
36                     //add and result store
37                     gen_add(tReg3, tReg1, tReg2);
38                     gen_st8(A, tReg3);
39                     //update condition codes structure
40                     gen_assemble_CC_param(LZE_ADD_OP,tReg1,tReg2,0x00);
41                     break;
42 ...
43 }
```

Listing 4.1: Decoding method code snippet.



$(PC) \leftarrow addr_{15-0}$

Figure 4.1: MCS-51 LJMP addr16 encoding and operation.



$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

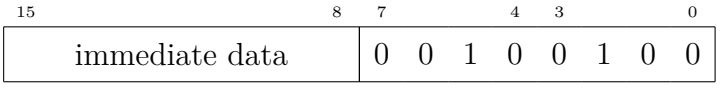
Figure 4.2: MCS-51 RET encoding and operation.

The expansion of the presented instructions to micro operations is now explained. The first step is to know the operands and the operation (behavior) of the source Instructions. Figures 4.1, 4.2 and 4.3 show the encoding and operation of the 8051 instructions `LJMP addr16`, `RET` and `ADD A, #imm8`, respectively. During the decoding, the remaining bytes of the operations, if any, are fetched and isolated from the source binaries.

For the `LJMP` operation, the two bytes address is fetched individually from the source binaries and merged into a 16-bit word, which is then used by the IR micro Operation `gen_writePC(uint16_t npc)`. This stores an immediate address to the source PC. Since `LJMP` is a branching operation, the end of a BB is assigned by setting the flag `eoBB` as true and the decoding ends.

The `RET` is a program branching instruction used to return the execution from a subroutine to the *caller* routine, which involves stack management operations (Figure 4.2). In order to streamline the operation reproduction, this instruction is emulated, by calling a helper functions. After concluding the decoding, a `gen_helper()` function is called. This function will generate the code necessary to call the `helper_ret()` C++ emulation function during Execution. This mechanism will be further detailed ahead.

The `ADD` instruction decoding starts with the fetching of the 8-bit immediate operand, stored at a temporary variable. Additional instructions are then performed, consisting in loading the value stored in the 8051 accumulator, loading



$(A) \leftarrow (A) + \#data$

Figure 4.3: MCS-51 ADD A, #data encoding and operation.

the immediate data to a temporary register, perform the sum and store the results back to the Accumulator. The IR micro operations used here are:

- `gen_ld8(uint8_t DReg, unsigned int imm);`
- `gen_movi(uint8_t DReg, uint32_t imm);`
- `gen_add(uint8_t DReg, uint8_t SReg1, uint8_t SReg2);`
- `gen_st8(unsigned int imm, uint8_t SReg);`

These micro operations were employed accordingly with the described operation of the instruction. There is one additional action not expressed in the instruction operation depicted in Figure 4.3, which is the condition codes update. This operation may only be performed during the execution of the instruction, since the CC values depend upon the result of the operation. To obtain this dynamic evaluation, a dedicated operation `gen_assemble_CC_param()` is used to preserve the operands and the operation on a dedicated structure during Execution. This structure will then be used by another helper function, in order to emulate the operation and update the correct CCs. This is the mechanism used to handle the CCs for every instruction that affects these flags, and will also be further detailed ahead in this chapter. The `decode()` might following other implementation algorithms, as long as the code generation is achieved through the use of the available micro operations, the architecture design is respected and the engineering effort is minimal, thanks to the adopted C++ strategy.

Source Complementary Memory

In order to support the source architecture's binaries execution, guest architectural features which hold information must be replicated. On what concerns data support features, since the information dynamically changes with the program execution there is a necessity to allocate the source memory in the target architecture. The MCS-51 architecture defines four distinct memories: internal RAM, SFR, program memory and external memory. The internal RAM is emulated in the source data memory, already present in the base DBT architecture, thus does not need any additional support. The SFR area, since is a memory mapped zone contiguous to the internal RAM, is also contained in the source data memory and dealt accordingly. The program memory is a write-only memory which is also intrinsically present in the DBT base architecture. However, the external memory

falls out of the base architecture design and additional support must be provided. The existence of such target memories was foreseen in the design and dedicated mechanism to handle them was provided. A pointer to 8-bit unsigned integer is declared in the derived class, and set to an allocated 8-bit array the size of the external memory. In this manner, the external memory is dealt with the same mechanism used to access the internal RAM, changing only the base address.

Helper Functions

As addressed above, the operation of some instructions might be too complex to translate to micro operations, originating extent IRs. In these cases, and adopting the QEMU strategy [86], helper functions are used to replicate the operation through a high-level language. A standard C++ method is added to the derived class, where the instruction operation is expressed in C++. This function is later invoked during Execution and executed, returning to the binaries executions after completion. In this subsection, only the decoding and operation reproduction is addressed. The calling mechanism, which is target architecture specific, is postponed to the next subsection 4.1.2, Target Specific Porting.

One of the MCS-51 instructions which requires helper functions usage is the Decimal-Adjust Accumulator (`DA`), portrayed in Figure 4.4. The CISC instruction applies to binary-coded decimal (BCD) representations of data and performs the adjustments to the result of two BCD packed variables. The instruction is decoded, similarly to the `RET` instruction, and uses a helper function, as shown in Listing 4.2.

```
1 ...
2 case 0xD4   : // DA
3     gen_helper(&CTranslator8051::helper_DA );
4     break;
5 ...
```

Listing 4.2: Helper function usage example.

The function `helper_DA(void)` is declared and coded in the derived class, so that its address can be passed as a parameter to the `gen_helper()` code generation function. The operation of the DA instruction is expressed in the `helper_DA(void)` C++ function presented in Listing 4.3.

The code is compiled and subject to compiler optimizations as part of the DBTor.

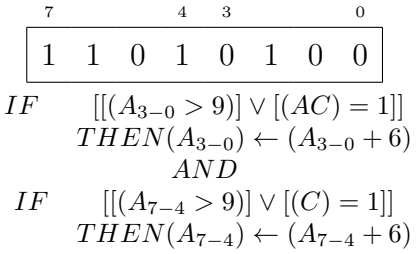


Figure 4.4: MCS-51 DA encoding and operation.

This mechanism offers the versatility of high level programing language when the existent micro operations fall limited in representing the instruction's operations, but may also be used to introduce tracing and profiling features in the translator.

```
1  void CTranslator8051::helper_DA(void){
2  //this helper function adjusts the eight-bit value in the Accumulator
3  //resulting from the earlier addition of two variables
4  //(each in packed-BCD format), producing two four-bit digits
5
6    int temp_Acc = env.dataMem[A];
7    int temp_PSW = env.dataMem[PSW];
8
9    //lower nibble adjust
10   if ( ((temp_Acc&0x0F) > 0x09) || ((temp_PSW&AC_BIT) != 0) )
11   { temp_Acc += 0x06; }
12
13   //higher nibble adjust
14   if ( ((temp_Acc&0xF0) > 0x90) || ((temp_PSW&CY_BIT) != 0) )
15   { temp_Acc += 0x60;  }
16
17   env.dataMem[A] = temp_Acc&0xFF;
18   env.dataMem[PSW] = (temp_PSW&~AC_BIT) | ((temp_Acc&AC_BIT >>8)<<
       CY_POS);
19 }
```

Listing 4.3: DA emulation helper function.

4.1.2 Target Specific Porting

The target specific porting efforts are in turn described in this subsection. The tasks are grouped in three general aspects, which are here detailed.

Dynamic Binary Translator Executable

One of the *sine qua non* conditions for DBT execution is the translation engine running on the host machine. This simple condition is provided by the compiler and host machine tool-chain. Despite this being a trivial aspect of the target support action, it should not be ignored, since is the compiler which confers the specificity to the translator of only executing in the designated host architecture. Helper functions transformation to target machine binaries is also attained through the compiler. The compiler is also considered a key element in the resourceability of the tool, since many mechanism are kept agnostic relying on the compiler actions (e.g., Translation to Execution transfer, helper functions, inheritance features). By the end of the source and target porting operations, is the host machine tool-chain which puts together all the different DBTor pieces, while correctly inter-operating.

Prologue and Epilogue Porting

As introduced before, a TBB is encapsulated between a prologue and an epilogue. These two elements ensure a smooth switch between the Translation and Execution domains, while ensuring an ABI compliant procedure call transition (AAPCS). This consideration leads to a porting decision regarding the working or temporary registers. The 3-address machine based IR must borrow working registers to the source architecture to execute each of the micro operation. These registers are assigned at the end of the decoding, when calling the respective micro operation function. The selection of the working registers is considered to be included on the target porting tasks, and must take into account the architecture's register utilization considerations, presented at the beginning of this section. In this case, three different approaches could be taken: (1) use only scratch registers and ensure register preservation during helper function calls, (2) use only preserved registers and ensure register preservation during Translation/Execution switching and (3) a mix of both. It was considered that register preservation operations executed during the prologue and epilogue of the TBBs would have less impact on the overall performance than saving and restoring operations prior and after helper function calls, thus the selected temporary/work registers used were the preserved registers, `R4-R11`.

Register `R4` was defined as the base pointer for the emulated source data memory and registers `R5`, `R6`, and `R7` were defined as the temporary registers number `tReg1`,

tReg2 and tReg3, as displayed in the code snippet from Listing 4.4.

```
1 //R4 to R11 are Preserved Registers
2 #define MEM_BASE 4  //data memory base pointer
3 #define tReg1    5  //temporary/working register #1
4 #define tReg2    6  //temporary/working register #2
5 #define tReg3    7  //temporary/working register #3
```

Listing 4.4: Working registers definition code snippet.

The code snippet in Listing 4.5 exhibits the code used to perform the prologue and epilogue operations. The prologue is executed with a push operation of the preserved registers that might be used during Execution. Additionally, the LR is saved to the stack, so the Execution's return address is not lost.

```
1  void CTranslator8051::gen_prolog(void){
2      //PUSH {LR | REG_LST} - save return address and all the
3      //working registers (preserved registers) to the stack
4      gen_PUSH( LST_LR | 1<<MEM_BASE | 1<<tReg1 | 1<<tReg2 | 1<<tReg3 );
5      //transfer parameter from (env.dataMem address) to MEM_BASE register
6      gen_mov( MEM_BASE, 0);
7  }
8
9  void CTranslator8051::gen_epilog(void){
10     //POP {PC | REG_LST} - restore all the preserved registers
11     //from th stack and return to the saved ink Register
12     gen_POP( LST_PC | 1<<MEM_BASE | 1<<tReg1 | 1<<tReg2 | 1<<tReg3  );
13 }
```

Listing 4.5: Prologue and epilogue decomposition into micro operations.

Finally, a gen_mov() micro operation is used to transfer the (int*)(env.dataMem), passed as a parameter from the Translation context through register R0 to the MEM_BASE defined register. The epilogue reverts the operations of the prologue, by restoring the original values of the preserved registers. Additionally the stack pop also transfers the stored LR value to the PC, thus returning the control to the Translation context and finishing the TBB execution.

Micro Operations Porting

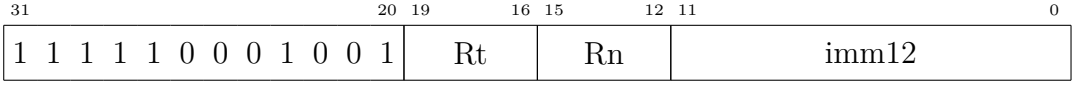
This task consists on generating the necessary code to reproduce the IR micro operations' functionality on the target architecture. Accordingly with the explained before, the use of an IR between the source and the target architectures enables

single-end porting, thus minimizing porting efforts. The back-end of this IR mechanism is the code generation by the DBTor's micro operations defined in the base Translator class as pure virtual methods and displayed in the Listing 3.1 of section 3.4, Deployment Insight, in Chapter 3. Here it is revealed the mechanism of how the functions-like IR generates code to the Tcache of the translator.

Each of the IR micro operations must be converted into target machine code, using the available assembly instructions of the target architecture [105]. As an example it is now shown the porting of the micro operations obtained from the decoding of the `ADD A, #imm8`, presented before on this chapter's section Source Specific Porting. The micro operations used are `gen_ld8()`, `gen_movi()`, `gen_add()` and `gen_st8()`.

The `gen_ld8()` must load an 8-bit value from the data memory to a work register. The `LDRB (immediate)` instruction from the Thumb-2 ISA [105], which calculates an address from a base register and an immediate offset and loads a byte into another register, was selected. The instruction has the encoding displayed in Figure 4.5. The operand `Rn`, is the base address of the load operation, the `imm12` is the offset from the base address, and finally `Rt` is the target register of the load operation. In the micro operation `gen_ld8()`, the parameters `DReg` and `imm` represent the destination register and the source memory address operands, respectively. These operands, together with the opcode `0xF890` (`0b111110001001`), are concatenated in a temporary 32-bit buffer accordingly with their role and instructions' encoding. The base address `Rn` is assumed by the `MEM_BASE` register. The micro operation porting ends with the transfer of the encoded instruction to the Tcache. The code snippet in Listing 4.6 expresses the described procedure.

The listings 4.7, 4.8 and 4.9 display the same procedure for the micro operations `gen_movi()`, `gen_add()` and `gen_st8()`, encoded based on the Thumb-2 instructions `MOV (immediate)`, `ADD (register)` and `STRB (immediate)` respectively (Figures 4.6, 4.7 and 4.8). The last micro operation porting presented is the `gen_helper()` (Listing 4.10), used to generate code that in turn will call auxiliary helper functions during the Execution.



LDRB <Rt>,[<Rn>,#<imm12>]

Figure 4.5: Thumb-2 LDRB (immediate) - Load Register Byte (immediate) encoding.

```
void CTranslator8051::gen_ld8(uint8_t DReg, unsigned int imm){
//LDRB (immediate) LDRB <Rt>,[<Rn>,#<imm12>]

    uint16_t tmp1, tmp2;
    tmp1 = (DReg << 12) | imm;
    tmp2 = 0xF890  | (MEM_BASE<<16);
    transCache.cacheCode(tmp2, tmp1);
}
```

Listing 4.6: Gen_ld8 micro operation code generation.

```
void CTranslator8051::gen_movi(uint8_t DReg, uint32_t imm){
 //MOV(immediate) MOVW <Rd>,#<imm16>

    uint16_t tmp1, tmp2;
    tmp1 = (imm & 0xFF) | (DReg << 8) | ((imm & 0x700)<<4);
    tmp2 = ((imm & 0x800) == 0) ?   0xF240 : 0xF640 ;
    tmp2 |= (imm >> 12);
    transCache.cacheCode(tmp2, tmp1);
}
```

Listing 4.7: Gen_movi micro operation code generation.

```
void CTranslator8051::gen_add(uint8_t DReg, uint8_t SReg1, uint8_t SReg2){
// ADD (register) ADD <Rd>,<Rn>,<Rm>

    uint16_t tmp1, tmp2;
    tmp1 = 0x0000 | (DReg<<8) | SReg1;
    tmp2 = 0xEB00 | SReg2;
    transCache.cacheCode(tmp2, tmp1);
}
```

Listing 4.8: Gen_add micro operation code generation.

```
void CTranslator8051::gen_st8(unsigned int imm, uint8_t SReg){
//STRB (immediate) STRB <Rt>,[<Rn>,#<imm12>]

    uint16_t tmp1, tmp2;
    tmp1 = imm | (SReg & 0xF)<<12;
    tmp2 = (MEM_BASE & 0xF) | (0xF88 << 4);
    transCache.cacheCode(tmp2, tmp1);
}
```

Listing 4.9: Gen_st8 micro operation code generation.

The parameter received is the address of the function to be called. Accordingly with the calling procedure explained at the head of this section, the class base

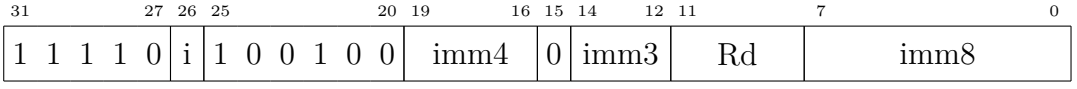
address is passed as the first and single parameter, thus it is generated code to move the base address `this` to `R0`. Then the destination function address is stored on a register (e.g., `R1`), and a branch and link operation to that location is generated. This mechanism emulates the environment function call, thus the compiler ensures the preservation of the work registers used in the execution.

```
1 void CTranslator8051::gen_helper(void (CTranslator8051::*i)(void)){
2     //generates the code for calling a helper function
3     gen_movi(0, (int)this);
4     gen_movi(1, *(int*)&i);
5     gen_blx(1);
6 }
```

Listing 4.10: Gen_helper micro operation code generation.

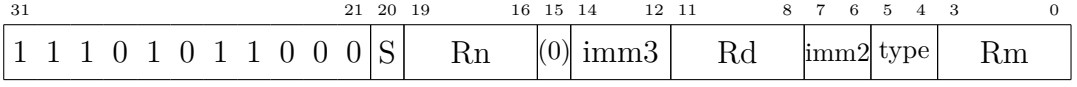
4.1.3 Condition Codes

The CC or conditional flags, are control and status bits, affected not only but mostly by arithmetic operations, that play an important role on the execution flow of a program. Conditional like "equal to", "greater than" or "smaller than" are inferred from the value of these bits in order to decide the outcome of condition branches. The CC often include bits that assign the occurrence of common condition such as a carry out, an arithmetic value overflow or a zero value on a determined register, but may also include some exotic flags that assign occurrences of carry out between byte nibbles or bit parity (even or odd). The type and number of conditional flags present on any architecture is strictly dependent of



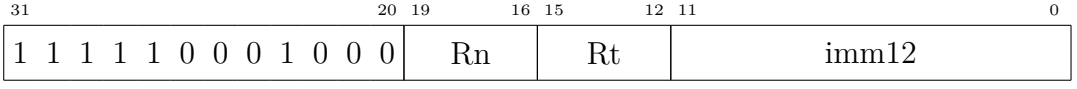
MOVW <Rd>,#<imm16>

Figure 4.6: Thumb-2 MOV (immediate) - Move (immediate) encoding.



ADD <Rd>,<Rn>,<Rm>

Figure 4.7: Thumb-2 ADD (register) encoding.



STRB<c>.W <Rt>,[<Rn>,#<imm12>]

Figure 4.8: Thumb-2 STRB (immediate) - Store Register Byte (immediate) encoding.

ISA design decisions, what originate distinctive CC sets for different architectures. The flags update is performed by hardware and its logic is tightly coupled with the architecture, thus there is no overhead in the native CCs calculations, and the update is performed as part of the operations that have effect on them. On DBT, the source architecture CC must be emulated for a correct execution of the binaries. However, because of the frequently CC disparity reality between source and target architectures, the target CC do not match the source CC, thus the CC values must be obtained through emulation. In the proposed source/target ISA pair, from their respective CCs presented in Figures 2.2 and 2.5, there are two flags in common: the carry (`C`) flag and the overflow flag (`OV`, `V`). However, and due to one architecture being of 8-bit and the other 32-bit, the flags setting conditions are not the same in the two architectures, so every CC bit must be emulated through software. The presented helper function mechanism applies well to evaluate the CCs during the code execution, by calling an external CC emulation function.

From the decoding of the `ADD A,#imm8` instruction presented in the Listing 4.1, the final operation `gen_assemble_CC_param()` prepares this CC emulation mechanism to be executed during code execution. The code snippet bellow, in Listing 4.11, displays the loading of the instructions operands and operation into the `CC_Parameter` structure, which is a data support variable used by the helper function in order to calculate the resulting CC of the executed source instruction. After loading the structure, a helper call is generated for the `helper_CC()`, where the flags' values are calculated and updated, accordingly with the type of operations and operands involved.

In the MCS-51 architecture, the CCs affectation is expressed in Table 4.1. The "X"

Table 4.1: MCS-51 CC flags affectation.

Instruction	Flag			Instruction	Flag		
	C	OV	AC		C	OV	AC
`ADD`	X	X	X	`CLR C`	0		
`ADDC`	X	X	X	`CPL C`	X		
`SUBB`	X	X	X	`ANL C,bit`	X		
`MUL`	0	X		`ANL C,/bit`	X		
`DIV`	0	X		`ORL C,bit`	X		
`DA`	X			`ORL C,/bit`	X		
`RRC`	X			`MOV C,bit`	X		
`RLC`	X			`CJNE`	X		
`SETB C`	1						

indicates that the flag's value will be defined according to the operation's result, a "0" and a "1" indicate the value is cleared or set with the instruction, respectively. The arithmetic instructions (`ADD`, `ADDC`, `SUBB`, `MUL`, `DIV`) perform the CC updating through the described helper function mechanism. The remaining instructions update the `C` flag by direct bit manipulation with micro operations.

```
1  void CTranslator8051::gen_assemble_CC_param(char operation, uint8_t
       Op1Reg,
2      uint8_t Op2Reg, uint8_t caryReg){
3      //T2 STRB<c>.W <Rt>,[<Rn>,#<imm12>]
4
5      uint16_t tmp1, tmp2;
6
7      //move the base structure address to auxiliary register
8      gen_movi(aReg1, (unsigned int)& CCParameters );
9      tmp1 = 0xF880 | aReg1;                        //STRB opcode
10
11     tmp2 = ( offsetof(lzEvParStruct, inputOp1)) | Op1Reg << 12;
12     transCache.cacheCode(tmp1, tmp2);             //operand 1 store
13
14     tmp2 = ( offsetof(lzEvParStruct, inputOp2)) | Op2Reg << 12;
15     transCache.cacheCode(tmp1, tmp2);             //operand 2 store
16
17     tmp2 = ( offsetof(lzEvParStruct, carryIn)) | caryReg << 12;
18     transCache.cacheCode(tmp1, tmp2);             //carry parameter store
19
20     gen_movi(aReg2, (unsigned)operation);         //operation store
21     tmp2 = ( offsetof(lzEvParStruct, operand)) | aReg2 << 12;
22     transCache.cacheCode(tmp1, tmp2);
23
24     gen_helper(&CTranslator8051::helper_CC);
25 }
```

Listing 4.11: Gen_assemble_CC_param code generation.

4.2 Tests and Results Discussion

To test the translator and the source and target architectures porting, the BEEBS benchmark suite presented on Chapter 2 was used. The binaries of the benchmarks were loaded to the Microsemi's SmartFusion2 flash memory and, one at the time, loaded into the Ccache and executed through the DBT engine. The execution was

timed through a 64-bit timer from the Cortex-M3 hard-core, clocked at the same speed as the SoC, 122 MHz. The timer is started before calling the `runDBT()` and stopped after returning from it. The tests were repeated for four different Tcache sizes: 4 KB, 8 KB, 16 KB and 32 KB. The Tcache minimum size must be enough to fit the largest TBB found during translation, due to the translator not supporting BB partitioning yet. It was determined experimentally that the biggest TBB was nearly 3200 bytes long, so the lowest Tcache size was set to 4 KB. The biggest cache size is limited by the system's available memory, excluding heap and C stack utilization and variables. On the test platform, the ceiling Tcache size was set to 32 KB. Besides, and considering that the translator targets low-budget embedded systems, it was considered that these four sizes were a good representation of the resources commonly offered by these platforms.

The execution cycles for the different cache sizes are presented in Table 4.2. The execution correctness was verified by comparison of the final values of the source and target emulated general purpose registers, SFR and memory. After performing the tests several times (>10 times) it was also verified that there is no cycles variation in the execution time. This observation was expected since there are no jitter sources on the developed demonstrator, nor on the test system.

For a better perception of the obtained results and the impact of the Tcache size variation on the execution, the graphic represented in Figure 4.9 was assembled. The graphic displays the ratio between the target execution clock cycles and the native execution clock cycles (presented in Table 2.1) of every benchmarks, for all the different Tcache sizes. Lower bars indicate smaller ratios between target

Table 4.2: BEEBS benchmarks results in clock cycles for different Tcache sizes.

	FDCT	2D FIR	CRC32	Float Matmul	Cubic root solver
4 KB	38427118	60104663	286594849	596952276	653674026
8 KB	27563054	58293771	272921832	570339030	622115667
16 KB	24741538	46255420	67105918	525301022	530945315
32 KB	21696723	25629587	67085400	206196708	474314975
	Integer Matmul	Dijkstra	Blowfish	Rjindael	SHA
4 KB	1624945813	2721839347	14373482249	15890039335	61797356085
8 KB	403797045	1562260959	7553385770	11653319425	39950737066
16 KB	279683259	900492708	8375191690	11479072783	25382638325
32 KB	309275719	716048566	12367325937	17299556826	36141117175

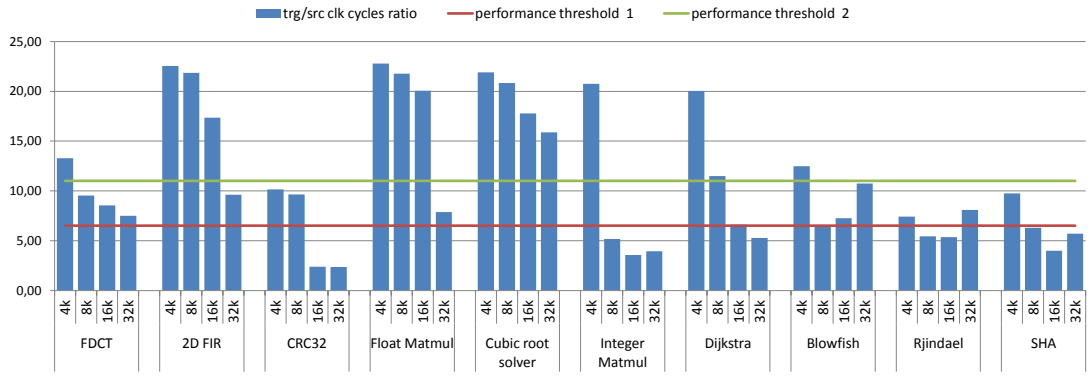


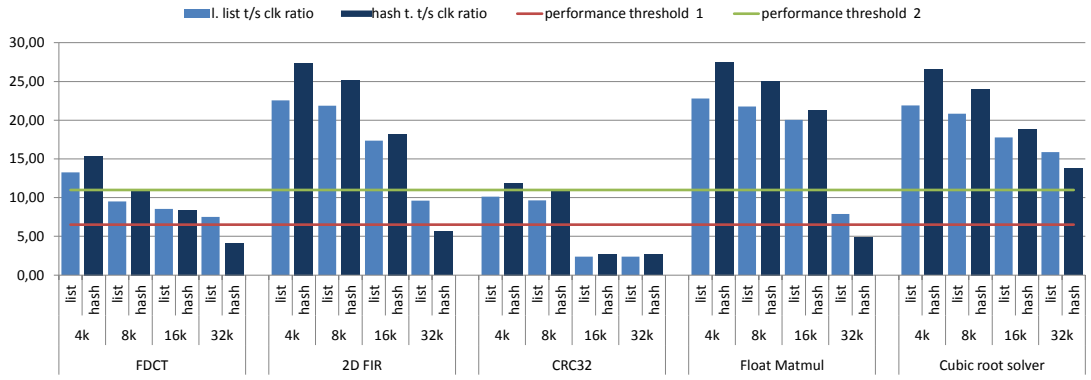
Figure 4.9: Target/source global execution ratio, for different Tcache sizes.

and source clocks, which invoke better performance. The ratio between the target and source execution clock cycles varies between approximately 23× (22.8, Float Matmul, 4 KB) and 2.5× (2.37, CRC32, 16 KB and 32 KB) slower. The results are uplifting, considering the minimalist approach followed (deployment without advanced decoding algorithms, unoptimized code generation, simplistic Tcache), the use of an IR for multiple source/target architectures bridging and the one-to-many instructions mapping. For a time based comparison, and on an speculative exercise, since this analysis is based on educated assumptions, the Cortex-M3 mainstream core line with a clock speed of 72 MHz was selected, considering the modern cortex-m3 processor's clock speeds variation from a few dozens MHz (32 MHz for the low-power families) up to a few hundred MHz (216 MHz in the powerful Cortex-M7). For the source clock speed, the common MCS-51 legacy cores used a 11.059 MHz clock frequency. This originates a target/source clock ratio of 6.51, represented in the same figure as the "performance threshold 1". Under these condition, 13 out of the total 40 tests would be running faster in the translation engine than on the native platform. Going even further on the conclusion, and taking the obtained deployment of the translator, running at 122 MHz on the Microsemi's SmartFusion2 Cortex-M3 core, the clock ratio of 11× is represented as a the "performance threshold 2" green line. Under these condition, 25× of the tests execute in less time under the translator than under native execution, representing 62.5% of the tests.

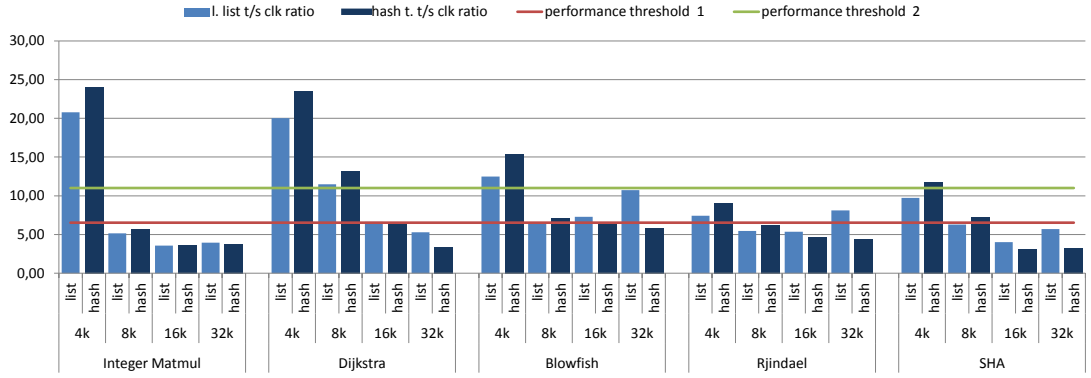
From this representation it is possible to conclude that Tcache size increase has a direct impact on the execution performance. A size increase most commonly causes a performance increase but the relation is not linear. There are even benchmarks where the Tcache size increase does not correlate with the performance gain. The fundamental fact for the performance increase is that a bigger Tcache accommo-

dates more translated code, thus fewer translations must be evicted in order to give place to newer translations. This causes more Tcache hits and less misses, reducing the issuing of repeated translations of code that was still in use but had to be discarded. There are tests where the Tcache variation has a sudden impact on the performance, such as the CRC32 (8 KB to 16 KB), Float Matmul (16 KB to 32 KB) and Integer Matmul (4 KB to 8 KB). This is due to occasional relations between the benchmarks' cycles size and the Tcache size, suggesting that the performance bumps occur when the Tcache size becomes big enough to accommodate the translations of the full source binaries, or at least an extensively executed cycle(s). This phenomena however does not explain why benchmarks such as the Blowfish, Rjindael and SHA do not consecutively reduce their execution time with the Tcache size increase. On these three cases, the performance improvement for a Tcache bigger than 4 KB, is clearly due to the size increase. However, the performance decrease, specially for the biggest Tcache size (32 KB), can not be explained with Tcache and TBB size relations. It was found that in these cases, what was deployed as a simpler and apparently effective solution, degrades the performance for greater Tcache sizes and large binaries whose translation do not fit completely in the Tcache. The cause is the Tcache search mechanism, deployed as a linked list and with insertion and access at the tail, for spatial locality of reference advantage purposes. The size of the TBB also plays a role in this assertion, because the smaller the TBBs are, the more TBBs will be accommodated in the Tcache, and the more extensive the linked lists becomes, increasing the search time for missed TBB prior to order a new translation.

An alternative Tcache management algorithm based on a "minimal and efficient" hash table macros (`uthash.h` [123]) was deployed, in order to confront both implementations and evaluate the impact of the constant-time search approach of the hash table method. It should be noted that the hash approach was initially dropped due to the expected excessive overheads associated with the hash key computation. Figure 4.10 add the benchmark results of the hash table managed Tcache to the already presented linked list managed Tcache, from Figure 4.9. It is now possible to compare both deployments and observe that none fully exceeds the other. The linked list management results show better performance for smaller Tcache sizes (4 KB and 8 KB), while for an intermediate 16 KB size there is no consensus. For the greater size (32 KB) the results for the Tcache managed by hash table always overpass the linked list results. Nonetheless, hash table managed Tcache tests perform consistently better with the Tcache capacity increments. The



(a) FDCT, 2D FIR, CRC32, Float Matmul and Cubic root solver results.



(b) Integer Matmul, Dijkstra, Blowfish, Rjindael and SHA results.

Figure 4.10: Target/source global execution ratio, for linked list and hash table Tcache managements for different Tcache sizes.

explanation rely on the fact that while the search time for the linked list managed Tcache varies with the number of TBB in the Tcache, in the hash table managed the search time is always the same. This leads to although the hash key computation overhead being considerable, it is exceeded by the search time when a large number of TBB are cached. This is the case for some of the 16 KB results and all of the 32 KB results, except when all the TBB fit into the Tcache.

Thus, hash tables have prejudicial impact on small Tcache sizes configurations of the DBT engine, because of the implied hash key algorithm computation, however this cost starts to pay off for greater Tcache sizes.

This conclusion suggests the implementation of hardware assisted Tcache management algorithms to improve the translated code management and access. Such hardware mechanism should have no performance harm on any Tcache size since insertion and search mechanisms should be performed in parallel with the software and deterministically. Another suggestion regarding the Tcache management is

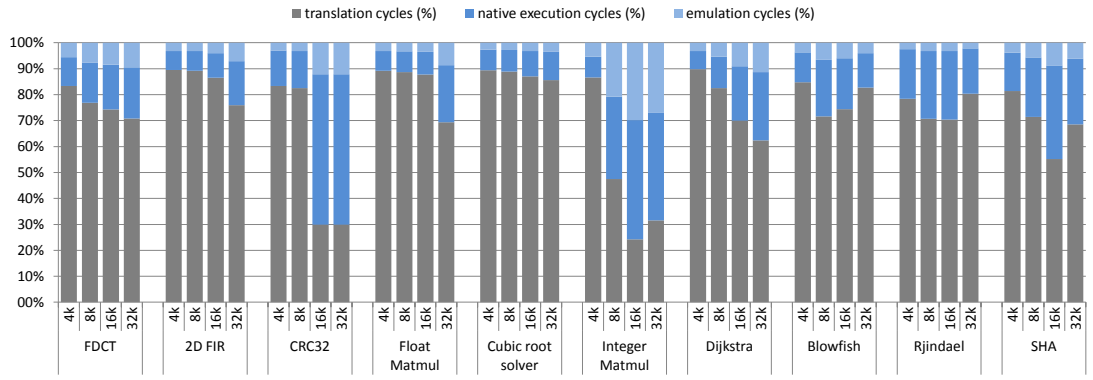


Figure 4.11: Global execution characterization in percentage, for different Tcache sizes.

studding the impact of different eviction mechanisms other than the full flush, currently used.

Considering that the Tcache sizes that mostly characterize the low-resource embedded systems are the more reduced ones (4 KB, 8 KB and 16 KB), and that the linked list Tcache management originated better results for these Tcache sizes, the remainder of this chapter focus on this management algorithm.

For a deeper analysis of the execution of the system, the global execution of the translator was characterized. Three aspects of the execution were discriminated: the translation and the execution, being the execution further categorized as the native execution, i.e., when translated code is being executed, and the emulation, which corresponds to the auxiliary helper functions execution. The results are presented in the Figure 4.11 and are expressed as a percentage of the total execution cycles, displayed in Table 4.2. From this graphic, it may be confirmed that with the Tcache size increase the translation parcel reduces, representing a smaller percentage of the total cycles spent in the execution. It should be noticed that the graphic is a percentual representation of the execution times, thus the difference in bar lengths for distinct Tcache sizes do not represent the same magnitude. The execution parcels (native and emulation) should remains constant, despite not being possible to draw such conclusion from this graphic. The emulated execution includes the execution of four helper functions: `helper_call`, `helper_ret`, `helper_DA` and `helper_CC`. This slice of execution represents an accountable part of the global clock cycles, specially on the math intensive tests, accounting up to 30% of the execution time, and 7% on average. Because of the CC emulation being present in every arithmetic operation, and due to the necessity of handling

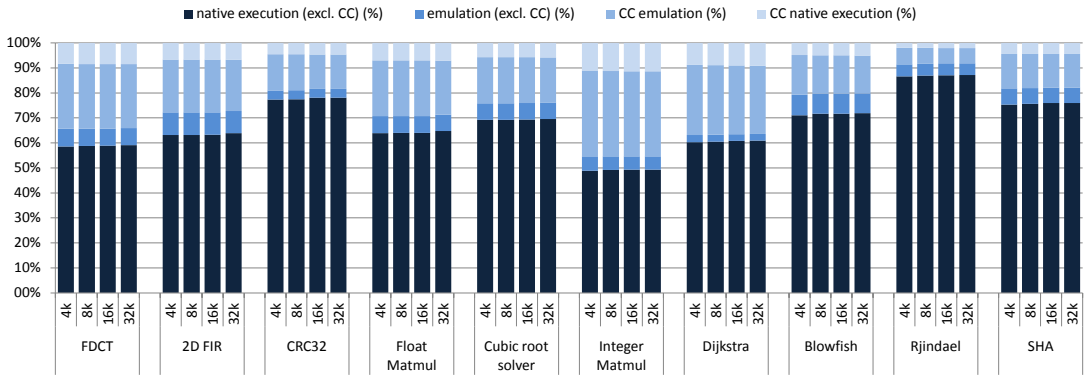


Figure 4.12: Execution characterization in percentage, for different Tcache sizes.

it though emulation, most of the emulation time corresponds to CC emulation.

The execution cycles were further characterized and studied, taking into account the clock cycles spent on CC handling. Because there is also native code associated with helper functions calling, and specially with the CC handling helper function, which must pass the operands and operation to the `CC_Parameter` structure, the native execution cycles dedicated to CC were accounted separately, as well as the cycles spent in the `CC_helper` function. The results are presented in Figure 4.12. The execution not involving any CC activity is represented in the two bottom colors, on the other hand, the CC related execution (native and emulation) are represented by the two lighter top bars. First, it was observed that there are no changes in the execution parcels for the different cache size tests, corroborating the findings extracted from Figure 4.11 analysis. This expresses that the execution is not affected by the number of translations nor by the translated code location. Second, it was also observed that the percentage of cycles dedicated to the CC calculation is on average 26% of the execution cycles, reaching up to 45% of the computation cycles on arithmetic intensive benchmarks (e.g., Integer Matmul). To globally characterize the clock cycles cost associated with the CC handling, and because the translation cycles spent in such handling were unknown, the graphic depicted in Figure 4.13 presents the total clock cycles cost associated with the CC emulation mechanism.

From this graphic it is possible to quantify the percentage of clock cycles dedicated to the CC handling during the binaries translation. Despite only being executed once for each CC affecting instruction, for smaller Tcache sizes, where translated TBBs are expected to be evicted to give room for other translations, the cost of code generation for the CCs adds up with repeated translations. This translation

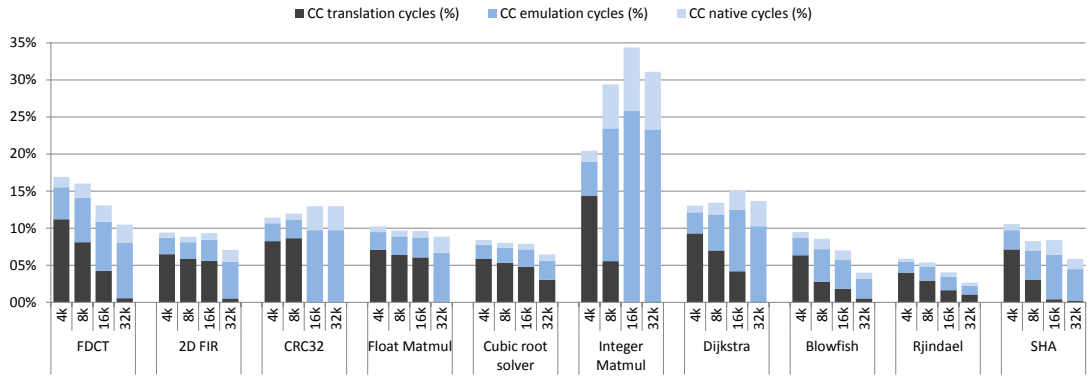


Figure 4.13: Total CC handling characterization in percentage, for different Tcache sizes.

cost even exceeds the execution costs of CCs handling for the smaller Tcache sizes for almost every test. The overall clock cycles dedicated to CC handling on the system varies from 2.7% to 34.4% of the tests total time, with an average of 11.5%. For the tests where was observed that the Tcache size was big enough to accommodate the full translation of the binaries (CRC32 16 KB and 32 KB; Float Matmul 32 KB; Integer Matmul 16 KB and 32 KB), the CC translation cycles become negligible (<0.05%). According to these results the handling of the CCs could and should be addressed as a performance harmful task, deserving the application of techniques to attenuate such overheads. Furthermore, there is no CC handling time in the source architectures, every time dedicated to it in the DBT process is already incurring in overhead. The same assumption may be made for every behavior emulated through helper functions, however the high frequency from which the CC emulation routine is executed makes it specially critical.

4.3 Conclusions

In this chapter was presented the demonstrator of the DBT architecture previously proposed. The source and target architecture porting tasks were described and exemplified with code listings. The requirements and design decisions proposed in the previous chapter were fulfilled, while the implementation techniques successfully ensured the separation of the DBT kernel, the source and the target concerns, promoting code reuse. The proposed IR successfully bridged the source and target instructions translation, while its employment on the DBTor grants ease of pairing for other source and target architectures porting. Despite func-

tionally and execution-wise correct, source and target porting is a fastidious work to be deployed manually, thus it is recommended the use of automation tools for automatic ISA pairing for future source and target porting.

The obtained DBTor is functional and executes all the 10 benchmarks from BEEBS benchmark suite, ensuring correct computation of the data, verified through the comparison of the final values of the source and target emulated general purpose registers, SFR and memory. It was also verified that there is no variation in the execution time. Performance-wise, the presented system required more cycles to perform the computations of the source binaries, taking advantage of the modern architectures higher clocks, it is possible to compete with the source execution time. Tcache size has great impact on the DBTor' execution. Generally a bigger Tcache originates better performance results. Ideally the Tcache should be big enough to fit the translation of the complete source binaries at once, however that is not possible in most of the execution cases, reason why Tcache eviction mechanisms are required. The Tcache management algorithm was also found to have impact on the performance. A solution totally or partially based in hardware would greatly reduce the overhead introduced by the TBB look-up.

Both the instruction decoding and the code generation mechanisms might be improved, contributing to reduce the translation time overhead. The execution time might also be improved through optimization algorithms and more efficient code generation methods. The use of such techniques imply overhead cycles to perform the optimization algorithms and a bigger memory footprint, with possible unpredictable results, since in order to become effective, the optimizations obtained must reduce an amount of overhead equal or greater than the overhead caused by the application of the optimization itself.

Relatively to the CC flags, extensive analysis was performed and presented, showing that its emulation has a significant impact on the DBTor performance. The CC update activity is performed by hardware and in parallel with the code execution, thus does not figure in the native execution time of the source binaries.

Hence, several improvement directions were identified: (1) the translation process, i.e., an improved decoding mechanism for quicker source instruction to IR micro operations decomposition; (2) generated code quality, i.e., adequate the generated code for the target architecture, through direct register mapping, TBB chaining and other light weight optimization techniques; (3) Tcache management, i.e., management mechanisms with reduced overhead; and (4) emulation mechanisms

efficiency, for a first pass code emulation and in CC emulation. The translation process and the generated code quality might be optimized based on known software techniques migration to the embedded arena. Regarding the Tcache management, it is suggested that it would greatly benefit from hardware acceleration. The topic (4) requires further investigation efforts, namely on the CC emulation, since less research is known to deal with this topic. In an attempt to improve the CC handling the next chapter will focus on optimization techniques to reduce the overhead of such emulation.

Chapter 5

Handling the Condition Codes

In this chapter are evaluated different CC handling methods deployed on the presented DBT engine. A novel technique to handle CC using COTS architectural debug hardware as a triggering mechanism is introduced, while assessing and comparing it with two existent CCs evaluation methods on the resource-constrained embedded systems arena. The proposed method is, functionality-wise, comparable with reconfigurable hardware modules or ISA extensions in open architectures. It is source architecture independent and also outcomes a limitation identified in one of the addressed techniques. The remainder of this chapter is organized as follows: Section 5.1 introduces and identifies the problem, Section 5.2 presents the methods used, on Section 5.3 are presented the evaluation tests and the results are discussed and finally, the conclusions are drawn in Section 5.4.

5.1 Introduction

Upon different source and target architectures, a DBT engine must deal with the correspondence of source and target ISA elements other than instructions, among other matters. In order to replicate the exact behavior of the source binaries execution into the target machine, the DBT engine must carefully emulate the ISA elements affecting the execution flow, namely, the Condition Codes (CC) or Flags Bits. Because of the inherent differences between computer architectures, i.e., the variation of the quantity and quality of the CC and how and when they are affected, CC emulation is a widely recognized challenge in DBT [16,37,45,99,124]. The CC of the source architecture must be emulated on the target architecture

in order to achieve the correct execution flow of the binaries under translation. Emulation through software routines provides flexibility in the implementation and ease of porting of the DBT.

Upon emulation, the source architecture's CC are calculated and/or updated by software when their condition is affected by source instructions. Due to the overhead that these operations generate and regarding that CC are frequently updated multiple times before being read, optimization techniques must be used. On Harmonia [45] the authors try to reduce the overhead associated using the CC using three techniques: (1) eliminating CC *redundant-compare* code, (2) running a CC liveness analysis and consequent dead CC elimination, and (3) through ISA extensions. The techniques proposed are based on the characteristics of an ARM to Intel Atom translator and can not be used on translators with flag disparity (e.g., parity and auxiliary carry flags are not common) and different architecture's word lengths (e.g., 8-bit architecture's overflow is not reproduced natively on an 32-bit architecture). Chao *et al.* [124] eliminate redundant flag computations inside translation blocks and use Lazy Evaluation (LE) for inter block CCs optimization, claiming 20% code size reduction and 40% performance gains. A LE of the CCs consists on saving the operands and the operations that affect the CCs on a run-time location upon the execution of affecting source instructions, as described in [86]. CC flags are then calculated upon request by CC-sensitive instructions, saving unnecessary calculations and time. However, the authors do not mention the source and target architectures used to collect the results, which might be too optimistic for architecture pairs with CC disparity. Digital FX! [16] also employs lazy evaluation of the CCs, but since this translator is directed towards x86 to Alpha architectures, it is able to use the technique very efficiently, taking advantage of how CCs are updated on x86 architecture. Yao *et al.* [99] approached the CCs problem through the use of reconfigurable hardware, reproducing the CCs structure on FPGA and ISA extensions to bridge the architectural gaps. The implementation details are not described by the authors and since the results of such solution are presented altogether with other hardware optimization features, the benefit or penalty of the technique is unknown. Although expected to be efficient, this approach can not be used in Commercial-Of-The-Shelf products (COTS), because (1) the cores have closed architectures and (2) all the logic is hard-wired. Some of the results presented in the literature may have benefited from the substantial computing power on the target architecture and convenient similarities between source to target architectures. They mostly rely on data-flow

analysis to reduce redundant computations, LE of the CC, ISA extensions and use of additional hardware on FPGA to accelerate the handling of CC flags. Custom ISA extensions and architectural modifications are not possible to use on closed architectures or on the standalone COTS. Hardware extensions are an option only when the deployment technology does include configurable logic such as FPGA, and in such cases the integrations must be compliant with the SoC architecture. Hence the results of the used techniques on resource-constrained COTS embedded systems are unknown.

It was also found a lacuna in the LE technique when CC are mapped on memory data space. When CCs are kept on memory-independent hardware registers, accesses are made through dedicated instructions, thus making CC manipulation detectable at translation time. On the other hand, when CC are memory mapped [125, 126], it is extremely hard to foresee CC accesses during translation time, in order to generate code capable of triggering a LE during execution. This means that on a memory mapped implementation every instruction capable of affecting the memory is a potential CC modifier, and that CCs must be up-to-date to any instruction that can read from the memory. Hence, it generates additional overhead in the detection of the instructions' memory address. Moreover, more specifically in the MCS-51 architecture case, the P bit is updated according with the current value of the ACC. Since the architecture is a one-register machine, it causes the ACC register to be intensively updated, causing additional CC emulation overhead, even using LE.

In this chapter it is proposed a novel solution for handling CCs on a DBTor without configurable hardware resources, while surpassing the LE limitations on data space mapped CC architectures. The LE approach was complemented with the hardware debug features available in the core to create an event-triggered lazy CC evaluation that was called *Debug Monitor based* LE. This new approach can be applied to all processors that integrates the ARM CoreSight debug and trace solution. It is believed that the use of hardware debug features in this manner is a pioneer approach, and since it uses hard-wired on-chip logic there is no hardware overhead. Hence, the contributions of this chapter to the state of art are: (1) an evaluation of four different CC handling methods on the resource-constrained embedded systems arena, (2) a novel technique to handle CC flags using COTS architectural debug hardware, by using (3) a triggering mechanism comparable with reconfigurable hardware modules or ISA extensions in open architectures and without hardware overhead; (3) heading also to solve the LE limitations on

architectures with memory mapped CC.

5.2 Condition Codes Evaluation

In the 8051 architecture the CC are kept in the PSW, at address `0xD0`. The PSW as an SFR is mapped on a "direct access only" memory space, hence detectable during translation time, but with additional decoding effort. The condition bits present in the PSW are: bit 7, CY; bit 6, AC; bit 2 OV; bit 0, P. The remaining 4 bits do have purpose in the architecture, but are not condition codes, thus not addressed here. Although the MCS-51's CY and OV flags are also present in ARMv7-M, since one is an 8-bit and the other a 32-bit architecture, they behave differently and one can not be used to mimic the other's behavior, thus requiring software emulation. As an example and considering the OV flag, while on the MCS-51 it is set when the result of a sum exceeds the value `0xFF`, on Cortex-M3 the same flag is only set if the result of the same operation exceeds `0xFFFF`. Regarding the P flag, it is an architectural legacy feature indicating the odd parity of the ACC. This flag is not available on modern architectures, and since it refers to the current value loaded in ACC it is only reproducible by emulation. Furthermore, for portability purposes, the emulation strategy should be kept, as mentioned in Chapter 3. The three different evaluation methods are presented below.

5.2.1 Standard CC Evaluation

This base strategy for dealing with CCs is to update their value upon the execution of an instruction affecting the CCs. As a base implementation, all the instructions that affect the conditional flags generate four additional target instructions that update a structure holding the operation type and the operands. This step is transversal to every CC evaluation method. In the standard method, for every CC-affecting instruction, additional code is generated to perform a function call to the CC update routine. The update routine reads the operation and operands previously stored and then calculates and updates the source CCs' memory location (PSW). In the code snippet of Listing 5.1 is shown how the CC update helper function call is generated, after the generation of the `CC_Parameter` structure update code. For the P flag, it should be calculated and updated for every-time a value is loaded into the ACC, introducing a great amount of overhead in the

systems. Since it is a feature rarely used in programs, it was not supported in the presented version of the DBTor in Chapter 4.

```
1 void CTranslator8051::gen_assemble_CC_param(char operation, uint8_t
      Op1Reg, uint8_t Op2Reg, uint8_t caryReg)
2 {
3     //...
4     #if CC_HANDLER == standard
5         gen_helper(&CTranslator8051::helper_CC);
6     #endif
7 }
```

Listing 5.1: CC update helper function generation.

To ensure legacy support execution correctness, as intended in this thesis, this possibility must be taken into account, specially upon the translation of legacy code, where binaries could be obtained through Assemblers. Thus, this evaluation of the CC flags includes the P bit update for full legacy support.

5.2.2 Traditional Lazy Evaluation

Contrarily to the standard approach which, regardless of the use, always calculates the CCs, a LE method was implemented. In this method, CCs-affecting instructions do not update the flags immediately, but the CCs-affected instructions do. For instance, an `ADDC` (add with carry) source instruction, upon translation will

```
1 ...
2 case 0x34   : // ADDC A, #immed
3     #if CC_HANDLER == trad_lz_ev
4         if ( lz_ev_need_to_update) {
5             gen_helper(&CTranslator8051::helper_CC_trad_lazyEv);
6             lz_ev_need_to_update = false;
7         }
8     #endif
9 ...
10    /* IR decomposition and code generation */
11 ...
12    //update lazy evaluation of condition codes register
13    gen_assemble_CC_param( LZE_ADDC_OP, tReg1, tReg2, tReg3);
14    #if CC_HANDLER == trad_lz_ev
15        lz_ev_need_to_update = true;
16    #endif
17    break;
```

```
18 ...
```

Listing 5.2: Lazy evaluation CC handling example code snippet.

have to generate code to perform the necessary function call to update the CCs, so that later the CY and the other flag bits can be read and used with coherency. The P flag will also be updated, together with the other CCs. A dataflow analysis is performed during translation, to reduce CC flags update calls, as in [124] and therefore unnecessary overhead. This dynamic analysis is kept to a minimal level for overhead containment reasons. A flag called `lz_ev_need_to_update` is set `true` after every instruction that affects the CC (through LE), meaning that the CC must be updated before being read next time. The instructions that read the CC (e.g., `ADDC`) will only generate code to update the CC in case the `lz_ev_need_to_update` flag is set `true`, otherwise (i.e., when `false`) the CC update code generation can be skipped, because the CC are up to date. In the code snippet bellow, the `ADDC` instruction is shown as an example of an instruction that both "affects" and "is affected" by the CC, when using the Traditional Lazy Evaluation. The full legacy support cost for this type of evaluation is additional CC evaluation helper function calls, since each instruction that affects the ACC will also set the `lz_ev_need_to_update` flag.

5.2.3 CC Lazy Evaluation Integrated with Debug Features

The limitations of the state of the art regarding CC mapped on memory data space, identified in the Section 3.1, are related with the possibility of every memory access instruction accessing and modifying the PSW, the address of the CC flags. This requires an enhanced decoding of the memory access instructions in order to identify the target memory address of the instruction, and perform the necessary operations if the address is `0xD0`, the PSW address.

To avoid the recurrent updates of the CC (and the associated performance penalty) and the additional decoding overhead of a LE approach it is proposed the idea of employing the debug capabilities of the Cortex-M3 to perform the detection of memory accesses to the PSW address. Since this hardware is included in the core and runs with total independence and parallelism from the core, its integration would provide the extra logic in need without extra hardware costs.

The CoreSight architecture support two debug modes: (1) a halting debug mode

where core execution is halted for probing and (2) the debug monitor mode, which triggers a debug exception handler routine to perform the necessary debug operations. In accordance, a possible use of the debug monitor mode is to watch for memory accesses (read, write or both) and trigger a debug monitor exception on those addresses under surveillance. Since CoreSight is the industry standard for debug and trace by ARM, available not only on ARM products but also as intellectual property (IP) for third party, this presented method is portable to all CoreSight compliant COTS.

This mechanism was applied to monitor the memory accesses and trigger an exception that updates the CC flags upon read accesses of the PSW emulated memory location. One of the comparators of the DWT module present in the target ARM Cortex-M3 was programmed to watch the memory read accesses on the source memory address `0xD0`. Then, from the debug monitor handler the CC update routine is called to calculate and update the value of the CC flags. The flags will be computed based on the LE information stored by the last CCs-affecting instruction. By doing so, when there is a pending update and CCs are required by an instruction, an exception is triggered and the flags' values are calculated prior to use, ensuring a correct emulation. The DWT comparator is configured to watch only read accesses since on write accesses the previous CCs will be overwritten. A dataflow analysis strategy is still performed in this method, by enabling and disabling the Debug Monitor exception in the same way as the `lz_ev_need_to_update` flag previously described. The code snippet bellow shows the code used to enable and disable the DWT and configuring the type of access trigger form.

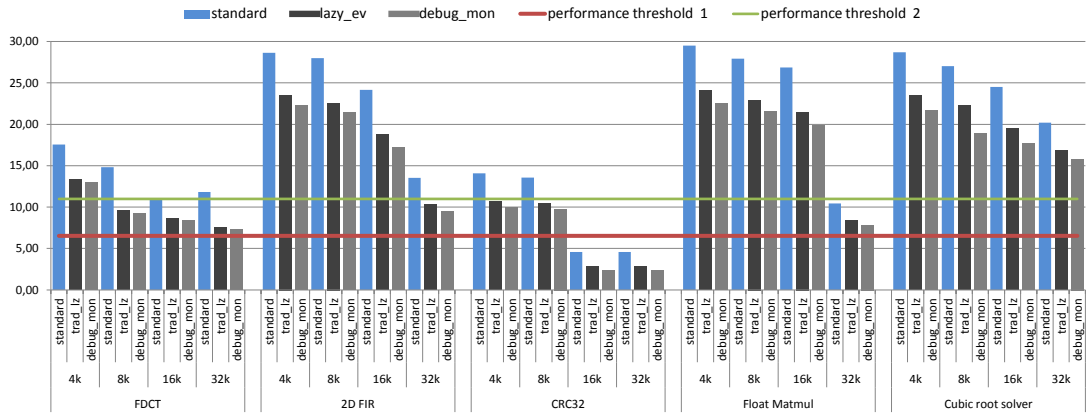
```
#if (CC_HANDLER == debug_mon )
    //MON_CMD values
    #define MON_RO  0x05
    #define MON_WO  0x06
    #define MON_OFF 0x00

    //macro to turn the debug monitor ON  or OFF
    #define set_debug_mon_CMP0( MON_CMD ) (DWT->FUNCTION0 = MON_CMD)
#endif
```

Listing 5.3: Debug monitor DWT comparator control code snippet.

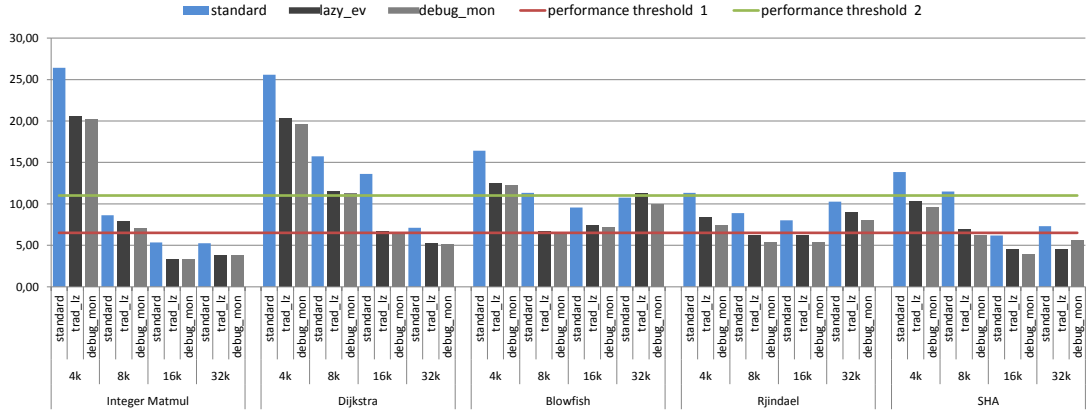
5.3 System Evaluation

The evaluation of the different approaches, Standard CC Evaluation, Traditional Lazy Evaluation and Debug Monitor based Lazy Evaluation, was carried through the execution of the BEEBS benchmarks for the four different Tcache sizes, 4 KB, 8 KB, 16 KB and 32 KB. The tests were performed for the linked list managed Tcache due to the better results that this management algorithm offers for smaller Tcache sizes and also because the smaller Tcache sizes will be the most common case in the resource-constrained embedded systems. The benchmark results are shown in the Graphics 5.1a and 5.1b in Figure 5.1, together with the performance threshold lines one and two, for reference.

The results display that, globally, a full legacy support, in detriment of an approx-



(a) FDCT, 2D FIR, CRC32, Float Matmul and Cubic root solver results.



(b) Integer Matmul, Dijkstra, Blowfish, Rjindael and SHA results.

Figure 5.1: Target/source global execution ratio, for the standard, lazy evaluation and debug monitor CC evaluation, using the linked list Tcache management, for different Tcache sizes.

imated computing approach of the CC flags, results in greater overhead. However the CC optimization techniques allow a reduction of that overhead to values close to the obtained with approximated computing in the previous chapter (Figure 4.9). In the experiments, traditional LE is consistently better than the standard method and the proposed novel method is also better than the traditional LE.

The *Traditional Lazy Evaluation* achieved a global overhead reduction of 24% on average. The greatest performance achievement was for the 16 KB Tcache size, with an average reduction of 28%, having the individual bechmark Dijkstra achieved a reduction of 51%. The novel debug monitor based approach achieved even better performance, contributing to a global average clock cycles reduction of 29.8%. The obtained average performance improvements per Tcache sizes are 26.5%, 31.2%, 34.3% and 27.3%, for the 4 KB, 8 KB, 16 KB and 32 KB, respectively. The variation on the results of the greatest Tcache sizes for the benchamrks in the Graphic 5.1b is explained due the performance penalty induced by the linked list management of the Tcache. Namely, in the case of the 32 KB Tcache SHA benchmark, it is noticeable a considerable performance penalty of the debug monitor methods over the lazy evaluation method. This is due to the debug monitor methods producing fewer target code (the CC handling routine calls are triggered, not embedded in the generated code), thus resulting in smaller BBs and causing a bigger number of BBs to fit in the Tcache. This sound consequence induces greater search time penalty in the linked list managed Tcache, thus the bad result of the benchmark.

There are also some variations in the amount of performance improvement, over the two optimized evaluation methods. Despite both methods implementing a lazy evaluation approach for the CC evaluation, they rely on different implementations and generate different types of overheads, thus originating distinct results. In other words, the debug monitor methods is capable of eliminating some of the extra decoding overhead through the migration of the CC update triggering functionalities to the CoreSight hardware, however it introduces the exception latency of the debug monitor exception. Hence, the variation of the benchmarks' size, the number of their intrinsic loops and its iterations have different impacts on the two lazy evaluation approaches.

The functional advantage of the *Debug Monitor based Lazy Evaluation* method for the problem of CC mapped in memory data space, approached on Section 5.1 is visible though the performance improvements of the benchmarks. Thus if

the CC of the source architecture are mapped into the data space [125, 126], the *Lazy Evaluation Integrated with Debug Features* reveals a valid contribution and should be used. This approach ensuring that the CC flags are only calculated when necessary, but are always up-to-date when accessed.

5.4 Conclusion

In this chapter, three different CC handling mechanisms in DBT specialized for embedded systems were assessed based on results obtained from actual implementations: (1) the standard evaluation of the CC upon the execution of CC-affecting instruction; (2) a traditional lazy evaluation of the CC; (3) a novel lazy evaluation based on the features of the CoreSight debug monitor hardware. The proposed new method proved to be functional and offered performance gains over the other two evaluated methods. Specifically, nearly 30% over the standard evaluation and 6% over the traditional lazy evaluation, on average.

The use of the debug monitor in the DBT architecture provides a triggering mechanism only comparable with the use of reconfigurable hardware modules in the architecture, but using only COTS features. Due to the CoreSight popularity this is a highly portable method. Its multipurpose and functionalities are appealing for peripherals mapping and exception emulation exploration also. The use of the debug features as a software triggering mechanism might be also applied in other type of applications than DBT, whenever a certain action is required upon dynamic and unpredictable behavior, e.g., code instrumentation, virtualization.

Regarding the CC evaluation optimization and acceleration, the results encourage hardware based approaches for overhead mitigation. This need is justified with the lack of parallelism that all the approaches failed to provide in the CC evaluation. Natively CC are updated with total parallelism from the program computations, but when DBT is applied, the CC emulation will compete against the source program computations for the processing resources, unquestionably resulting in overhead. However, in order to be COTS compliant, the hardware integration must come in a non-intrusive manner (i.e., without architectural modifications or ISA extensions), so that it can be considered as an acceleration solution for the low-cost/resource-constrained embedded systems.

Chapter 6

Non-intrusive Hardware-Assisted Acceleration

In the previous chapters an approach to DBT on resource-constrained embedded systems has been explored and evaluated, leading to several acceleration and improvement suggestions that rely on dedicated hardware. Software to hardware offloading is a common acceleration procedure used when software-only approaches do not meet the performance requirements. There are however other motivations to embark on software to hardware tasks migration, namely, functionality extensions and shortfalls overcome.

This chapter approaches hardware offloading to address the previously identified limitations of the DBT engine regarding the Tcache management and the CCs evaluation methods. The suggested approaches remain non-intrusive to the target architecture, which cope with the COTS-driven deployment of DBT for the resource-constrained embedded devices. It is proposed a Tcache management module and a non-intrusive hybrid acceleration architecture that extends the functionalities of the DBT engine to support peripherals remapping and to deal with fully dynamic behaviors, such as interrupt handling.

The remaining of the chapter is organized as follows: Section 6.1 presents a comprehensive introduction and the related work; Section 6.2 introduces and evaluates a hardware Tcache management mechanism. Section 6.3 proposes a non-intrusive acceleration architecture, while the remaining sections 6.4, 6.5 and 6.6, use the hardware based acceleration architecture to evaluate another CC handling mechanism, peripherals support and remapping, and interrupt handling, respectively.

Section 6.7 contains the conclusions drawn during the chapter.

6.1 Introduction

Based on evidences gathered through previous chapters, some features of the implemented DBT engine would probably benefit from hardware acceleration. It is known that software to hardware task migration is a performance enhancement technique widely used in many fields and applications, when software-only approaches do not deliver the necessary metrics; and DBT is not an exception.

In [127], Borin *et al.* presented a strategy to identify the main sources of overhead involved in the process of DBT. Despite the study concerning DBT aimed at desktop processors, thus, not applied directly to embedded systems, evaluation parameters such as cold code translation and translated code execution are transversal to the DBT topic. Among other suggestions, the authors point that research in overhead reduction through hardware support should be conducted in order to achieve near zero overhead DBT. Yao *et al.* [99] identified that common DBT systems suffer performance loss because of architectural heterogeneity among ISAs, control flow and context switches. In the same work, it is proposed an FPGA-based hardware/software co-designed acceleration solution, achieved through (1) register replication in reconfigurable hardware and (2) ISA extensions. The authors claim a global speed-up of 56.1%, but provide very shallow details on the integration of techniques and DBT engine characteristics. Despite apparently efficient, the approach comes with the cost of architectural modifications to the target processor. Gomes *et al.* [128, 129] have used hardware to deploy functionalities to solve problems found in real-time operating system (RTOS) state of the art. Through hardware assistance, the authors managed to unify the priority space in interrupt handling and also deployed a mixed hardware/software multi-thread scheduler for RTOS, increasing its predictability. The hardware extensions were integrated in the core in a tightly-coupled fashion, requiring access to the core architecture. This is considered a non COTS friendly approach of hardware integration, thus does not comply with the constrains of this work. In [130], graphics processing unit (GPU) integration in a translation framework is suggested as a form of exploring data-level and task-level acceleration opportunities from CUDA C compiled programs in AMD's runtime environment for GPUs. Although a heterogeneous solution would be viable in the deployed DBTor, the described work applies only for platforms

where a GPU is present, i.e., a high-performance system, which is not target of this thesis. DBTIM is a hardware assisted architecture-DBT for full virtualization [30]. The solution targets high-performance systems and uses a reconfigurable DBT chip, deployed in a DIMM, coupled into a motherboard in order to provide full hardware virtualization of the host CPU. The DBT chip receives the source code and the translation request through the memory interface, processes the request and delivers the translated code through the same mechanism. The approach is an example of hardware integration over traditional systems, without requiring architectural modifications to the target architecture. The use of FPGA fabric to promote binary compatibility is advocated in [131] and in [93]. The former proposes the use of hardware to promote binary compatibility, using reconfigurable coarse grained units to execute legacy functionalities through Dynamic Instruction Merging technique, a form of BT in hardware. In [93] the authors attempt to understand the challenges of applying reconfigurable computing to accelerate dynamic binary translation during runtime, using co-processors. The method is based in the detection of execution patterns in the source code upon its profiling and subsequent loading of accelerator modules bitstream to the FPGA.

In Chapter 4, Tcache management was pointed as a functionality that would benefit from hardware acceleration to solve the drawbacks found in both of the attempted management implementations. While the linked list management approach is associated with a lower search algorithm overhead than the hash table approach, with the increase of the list nodes, the search time exceeds the hash key computation time and the hash table mechanism performs better. In order to overcome the software-based approaches, the hardware assisted mechanism should ensure that (1) the insertion and search time of a TBB is reduced and that (2) the search time remains constant regardless of the number of TBB stored in the Tcache. Tcache management in DBT has already been approached in the literature. Baiocchi *et al.* in [61] use SPM as a auxiliary memory for quick context switch between the translation and execution environments; by reducing the translated code and by delegating code caching operations in the SPM. However, in this thesis' approach, the context switch is extremely reduced (one instruction to save the return address and another to move the data memory base address). Furthermore, SPM, or Tightly Coupled Memory (TCM), is not commonly present in the resource-constrained low-budget embedded devices. In [132] the authors resort to hardware techniques in order to manage the code cache either in DBT or dynamic optimizers. Hazelwood *et al.* [133] identify and study the Tcache performance on DBT,

presenting a framework to access and manipulate the translation cache of the PIN binary instrumentation system. Chen *et al.* developed extensive work in this topic [31,33,69,94], suggesting hardware to assist a specialized instruction decode cache, the DICache. Despite the work presenting substantial improvements over software-only deployments, it proposes the integration of hardware extensions at architectural level into an IP ARM processor core. Following Baiocchi, Yao *et al.* [99] also integrate SPM in FPGA to reduce context switching overheads. Furthermore, they present a hardware deployment of the mechanism proposed in [132], as a simple look-up table composed by a content-addressable memory (CAM) and a RAM. This mechanism seems to fit to the application scenario proposed in this thesis, but it still uses a software hash table as a secondary mechanism to decide if the TBB is cached or not.

In Chapter 5, acceleration approaches using hardware were attempted. However, considering the non-reconfigurable COTS devices targeted, the usable hardware was static and set available by the manufacturer, with a predefined functionality (not for acceleration purposes) that was exploited for the purpose of DBT enhancement. This initial approach using the debug monitor features of the CoreSight architecture has revealed neat, minimalistic and cheap to implement, because (1) it is compatible with fully dynamic behavior of the execution (contrarily to translation time approaches); (2) is lazy evaluation implementation friendly, once the necessary actions are only taken upon a read or write request (avoiding unnecessary updates); (3) introduces just small modifications to the DBT execution flow; (4) does not need a strong emulation environment; and (5) may be deployed on a wide range of COTS ARM processors. However, this implementation suffers from some drawbacks: (1) it does not co-exist with debug functionalities; (2) it is limited to the number of DWT comparators available in the device (commonly four); and (3) it requires the manipulation of the stack in order to modify the debug monitor interrupt service routine (ISR) return address to repeat the instruction that caused the interrupt. Again, Yao *et al.* [99] propose hardware assisted CC evaluation, but using flags register replication on FPGA. However the authors fail into provide information on how the CCs in hardware eliminates the CCs computations overhead through the native ALU related instructions.

The mechanism's potential for peripherals support was also suggested in the last chapter. Peripherals are undeniably important features of microprocessors that not only establish an interface between the microprocessor and the outside world but also enable the system to perform tasks in parallel with the processor's compu-

tations. Any legacy binary code will inevitably include access to at least a few peripherals (e.g., IO ports, UART, I2C, SPI, timers, analog-to-digital and vice-versa converters), thus becoming mandatory their support by the DBTor. Despite the proposed approaches, peripherals access is still identified as a bottleneck feature in DBT. In [95] the authors identify the peripherals as one of the main components of an embedded system. It is also discussed that static peripherals address detection, or even during translation time, is not straightforward to perform. Instead of goings through a table with peripherals location addresses during the decoding of load and store instructions, the authors use hardware support for address translation and look-up overhead reduction. Then, each access is catched by a peripheral simulator in order to replicate the peripheral actions.

Another mandatory feature that should be considered is the interrupt support. Interrupts are employed in many activities in embedded systems programing, such as inputs registration (I/O and capture mode interrupts), time event assignment (timer interrupts) or to notify the end of a task (serial ports interrupts). Regardless of the trigger source, the handling of the different interrupts relies on the same mechanism and remains fairly consistent between different architectures. After an interrupt being triggered, it is registered and, after priorities and other similar considerations, code execution is suspended and the designated handling routine is set to execute and attend the pending requested. The main aspect that differentiates an interrupt handling from regular and sequential code execution is that interrupts are unpredictable in time. This dynamic behavior introduces severe difficulties in the emulation process. The state of the art use periodic verification to emulate the source environment and detect pending interrupts. In QEMU [86], for performance purposes, interrupts are not checked at every TBB execution. Instead, a user call function verifies if pending interrupts must be evoked and then, interrupts are handled in the main loop of the DBT. Kondoh *et al.* [95] also follow a similar approach, using the peripherals simulators after detecting the interrupt request. In [128], Gomes *et al.* use hardware to solve an interrupt priority space problem in RTOS in a way defined by authors as "minimally intrusive at hardware architecture level". Nevertheless, their approach still relies on modules tightly-coupled to the processor's micro-architecture, which is not possible to perform in closed architectures.

To surpass the limitations identified over this section and to provide functionality enhancement, along with COTS products utilization, reconfigurable hardware must be serviced and integrated in the design. This precondition implies the use

of FPGA fabric to host the acceleration support system as well as the eventual acceleration modules. As foreseen in Chapter 2, the modern SoC solutions which integrate a hard-core processor and reconfigurable FPGA in the same device provide an excellent deployment environment. Although reconfigurable, the device still complies with the COTS requirement of this work, hence the integration of such accelerators must follow the available standard interfaces, leaving processor's micro-architecture unaltered. The development and integration of hardware accelerators should be sustained by an architecture that promotes flexible and scalable acceleration while minimizing the NRE efforts of individual accelerators integration.

This chapter presents a hardware based mechanisms to accelerate and extend the functionalities of DBT applied to the resource-constrained embedded devices. The mechanisms must be compliant with COTS deployments, thus an architecturally non-intrusive approach is desired. While the Tcache management is a DBTor-specific acceleration feature (source and target architecturally independent), the remaining accelerations possibilities suggested are dependent from the source architecture and its features to be emulated, thus the hardware acceleration should be deployed in a scalable and flexible way. Based on the limitations found during the CC acceleration through the debug monitor hardware features, the following characteristics were identified:

1. Flexibility - the same method should be applicable to multiple scenarios, e.g., CC handling, peripherals emulation, interrupt support, etc. This requires the solution to not be focused only on the feature to be supported, but rather on the interaction mechanisms involved.

2. Scalability - the method should be able to efficiently deal with multiple accelerators/extensions in the design.

3. Non-intrusiveness - to cope with COTS products the solution must not require architectural modifications at the processor level; and for ease of integration in the system the accelerators should not interfere with the DBT execution flow.

In the next sections non-intrusive hardware based solutions to address the issues described in this introduction will be presented and discussed.

6.2 Tcache Acceleration Assisted by Hardware

On an effort to speed up the translated code management (i.e., adding and searching translation entries) and obtaining a scalable management method, the Tcache management effort is delegated to hardware through an FPGA deployed solution. The approach followed in [99] is well suited to be applied because it is non-intrusive and takes full benefit of the hardware, but it still relies on a heavy software backup mechanism to handle missing translation from the hardware management. The software hash table mechanism should be avoided in the resource-constrained embedded systems, because of the already identified latencies it originates.

A full hardware Tcache approach was considered, but issues were faced regarding its implementation. A full hardware cache is a straight-forward implementation, either following a fully-associative, direct-mapped or set-associative policy [134]. However a Tcache does not exactly resembles a cache, but rather a buffer. While a typical cache stores a determined memory position, which the size is known and equal to every entry, a Tcache does not contain the equivalent representation of the source code at every address. Each translation is only addressable by its entry address and its size is variable and unknown upon its creation. Dealing with such traits in hardware would involve a complicated mechanism to manage the available space, and would probably result in small improvements, since most of the Tcache associated overhead is related with the management rather than the data caching and accesses.

6.2.1 Proposed Solution

An approach similar to Yao *et al.* was followed, however with some adjustment to avoid the use of software hash tables. The TBB caching, the eviction policy and the available space management are processed in the same manner as before (i.e., using a software approach), while adding new TBB entries and searching for a TBB are processed by auxiliary hardware. The hardware implementation is based on the regular Tcache memory space allocated on the target data memory plus a look-up table, similar to a fully associative cache (CAM) + RAM) on a integrated implementation, as depicted in Figure 6.1.

In this approach, and diverting from the one presented in [99], the output of the CAM is not returned to the microprocessor. In case of a source address *hit*,

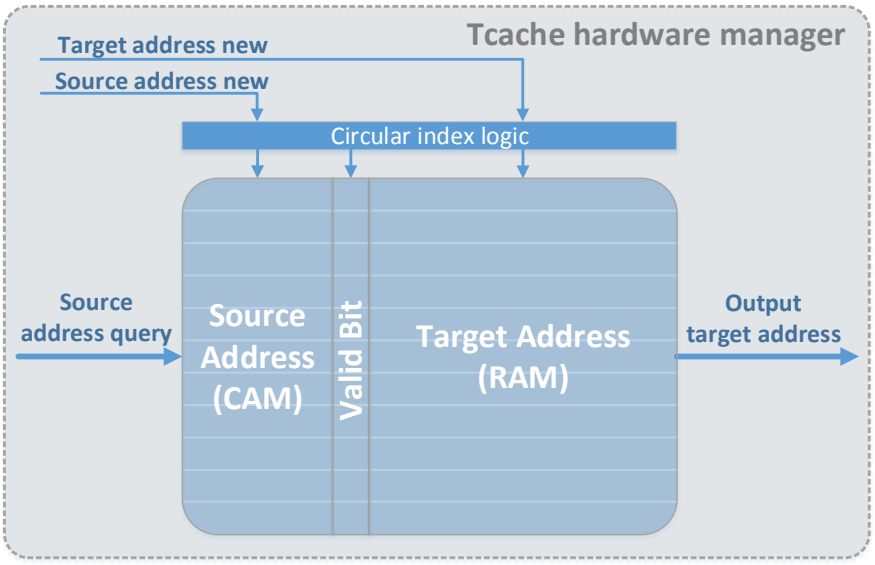


Figure 6.1: Tcache hardware manager diagram.

the target address is directly forwarded to the microprocessor. If an address *miss* occurs, then the address `0x00000000` is returned, indicating that the source address is not yet translated. This `NULL` address may be used in case of a *miss* because the Tcache memory is never allocated at the bottom of the source data memory. There is one additional valid bit in the architecture to prevent false *hits* during the first accesses and after Tcache resets. Although the number of TBBs that fit into the Tcache memory space varies, the number of the hardware look-up-table entries is fixed. To deal with this, other approaches include a software hash table to index additional TBBs after the hardware look-up-table being full. This induces the penalty of calculating a hash key and searching the hash table every time a Tcache *miss* occurs, even when the Tcache is not full. The implemented approach avoids the use of the software hash table through an adjustment to the new entry insertion mechanism. It is implemented resembling a circular list. The new entries are inserted sequentially until the insertion index overflows and starts to overwrite the first entries. The eventual look-ups of overwritten entries will generate a false *miss* because the translation is indeed stored at the Tcache allocated memory, but its look-up position was given to another TBBs, in favor of using the remaining Tcache allocated memory before requiring a full eviction. Moreover, older entries are less likely to be required. The hardware look-up-table size is modifiable through a parameter and should adequate to the Tcache size, which translates into the typical number of TBB per Tcache filling. From the ten benchmarks used it was empirically determined the following correspondence:

- 32K - 256 entries
- 16K - 128 entries
- 8K - 64 entries
- 4K - 32 entries

The hardware look-up is performed in one clock cycle, plus the bus access latencies, which represent a total of 5 clock cycles. The access time is deterministic and therefore remains constant, no matter the number of TBBs in cache.

6.2.2 Interface and Software API

The hardware-managed hybrid Tcache is seamlessly integrated in the DBT engine, in accordance with the followed OO paradigm. The interface methods remain the same as the linked list and the hash table approaches. The peripheral is connected through the standard AMBA 3 AHB-Lite bus [135] and mapped in memory through the register interface depicted in Figure 6.2.

A query on a source register is performed by writing the source program address to the TCACHE_SOURCE_ADDR_QRY register and reading the TCACHE_-TARGET_ADDR_GET register. The read value is the target memory address where the translation is stored, or a `NULL` address (`0x00000000`) in case of the queried BB not being translated. To add a new TBB entry to the look-up-table, the source PC address is written to the TCACHE_SOURCE_ADDR_NEW register, followed by a write of the target code location address of the TBB to the register TCACHE_TARGET_ADDR_NEW.

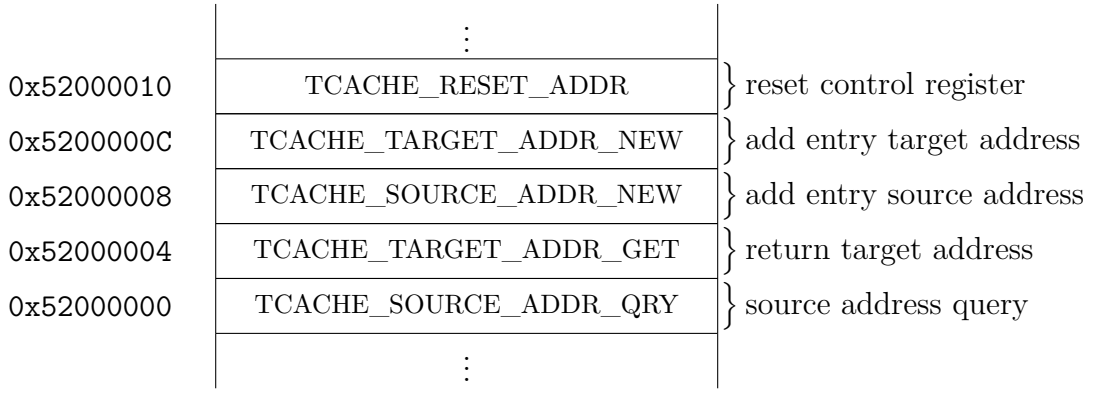


Figure 6.2: Tcache hardware manager peripheral mapping.

```
#define TCACHE_MAN_BASE          0x52000000U
#define TCACHE_SOURCE_ADDR_QRY   (TCACHE_MAN_BASE + 0x00)
#define TCACHE_TARGET_ADDR_GET   (TCACHE_MAN_BASE + 0x04)
#define TCACHE_SOURCE_ADDR_NEW   (TCACHE_MAN_BASE + 0x08)
#define TCACHE_TARGET_ADDR_NEW   (TCACHE_MAN_BASE + 0x0c)
#define TCACHE_RESET_ADDR        (TCACHE_MAN_BASE + 0x10)

#define HW_MANAGER_RESET()  *(int *)TCACHE_RESET_ADDR = 0x00000000

#define HW_MANAGER_GET_TRANS(QRY_ADDR, RET_ADDR) \
    *(int *)TCACHE_SOURCE_ADDR_QRY = QRY_ADDR;\
    *RET_ADDR = (unsigned char*)*(int *)TCACHE_TARGET_ADDR_GET

#define HW_MANAGER_NEW_TRANS( NEW_SOURCE , NEW_TARGET ) \
    *(int *) TCACHE_SOURCE_ADDR_NEW = NEW_SOURCE;\
    *(int *) TCACHE_TARGET_ADDR_NEW = NEW_TARGET
```

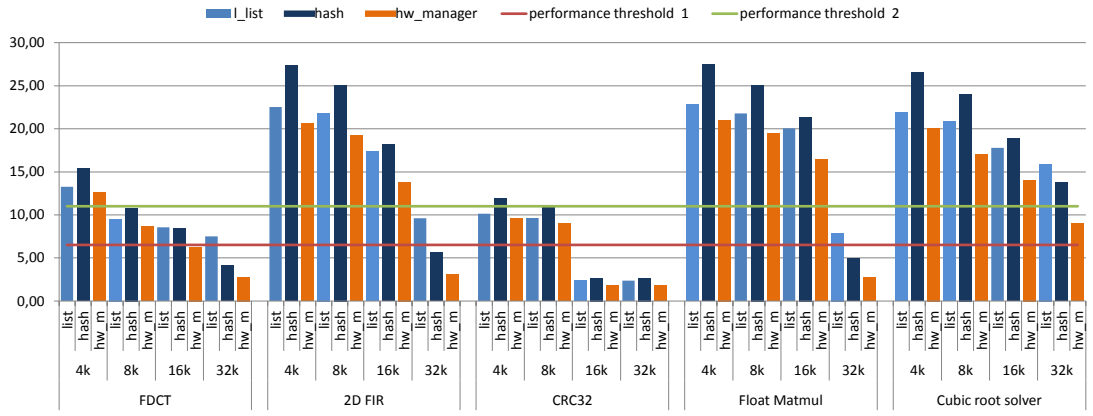
Listing 6.1: Tcache hardware manager driver.

Register TCACHE_RESET_ADDR is used to perform a reset to the look-up-table, via any written value. This mechanism is used to perform the cache evictions.

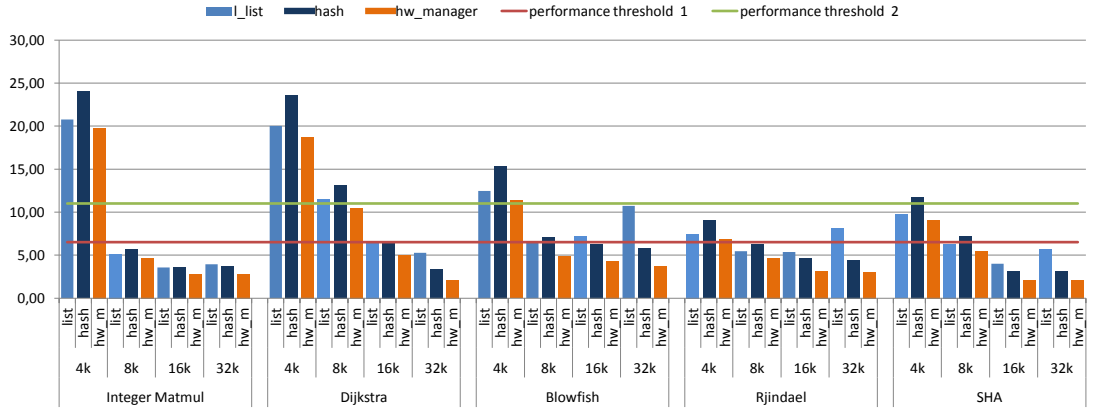
A simple device driver, partially presented in the Listing 6.1, was created to perform the queries, add entries and reset actions. The device driver is made out of simple macro expression, used to abstract the registers addresses and the interface sequence.

The new approach was evaluated using the same method used to compare the linked list and hash table Tcache management approaches, with the standard CC evaluation for the non full legacy support version of the translator. The graphics in Figure 6.3 aggregate the information from Figure 4.10 with the orange-marked column results of the hardware approach Tcache.

The new approach' performance exceeds both of the previously presented management techniques in every test, either for short or long programs, with big or small Tcache sizes. The hardware look-up-table results in faster management than the simple linked list approach and does not show the performance degradation on the greater Tcache sizes, observed in the latter approach. Furthermore, since its search time remains constant regardless of the number of TBBs, it outperforms the hash table management on the longer programs for the greater Tcache sizes. The results are on average 25% and 26% better than the linked list and hash table



(a) FDCT, 2D FIR, CRC32, Float Matmul and Cubic root solver results.



(b) Integer Matmul, Dijkstra, Blowfish, Rjindael and SHA results.

Figure 6.3: Target/source global execution ratio, for linked list, hash table and hardware based Tcache managements for different Tcache sizes.

implementations, respectively, with the highest performance increase for the linked list approach with a 32Kb Tcache.

Table 6.1 shows the synthesis results on the SmartFusion2 FPGA technology, for the different suggested entries count, in terms of FPGA resources (4-input Look-Up Table (4LUT), and D-type flip-flop (DFF)).

Table 6.1: Tcache hardware manager FPGA resource utilization.

Entries	**4LUT**	**DFF**	**Combined Resources**
32	1357	2109	3466
64	2704	3678	6382
128	5398	6816	12214
256	10785	13092	23877

6.3 Non-Intrusive DBT Acceleration Module

To introduce hardware acceleration into DBT and complying with the five premises identified in Section 6.1, a DBT acceleration architecture is proposed and its integration tested and evaluated in the following sections. Following the debug monitor hardware approach, a similar hardware module is suggested. By design, this type of mechanism already addresses the (1) flexibility and (3) non-intrusiveness premises from Section 6.1. Regarding (1) flexibility, the mechanism acts by detecting read and/or write accesses to configurable address locations, triggering the execution of diverse actions. Hence, it may be applied in multiple scenarios, from CC handling, to detect peripherals access. On (3) non-intrusiveness concerns, the method does not require any architectural modification neither ISA extension since it operates on regular architecture functionalities: address accesses and interrupt/exception handling. To address the (2) scalability problem, the proposed approach should deliver an adequate number of address comparators, which is possible to achieve by taking benefit from the FPGA reconfigurability.

6.3.1 Architecture

The proposed architecture functionality relies on a central module which is responsible for sniffing bus accesses and addresses, thus hereafter referred to as "sniffer". This sniffer module is placed in between the MSS and the data memory, in order to sniff read and write accesses to configurable memory addresses. Due to this integration, the MSS is not aware of the modules, on what concerns memory accesses, neither requires modification of the existent hardware or architecture. It is also integration-friendly because it allows easy functionality addition to the system without modification on the DBT kernel. This approach, contrary to the state of the art, does not requires any ISA extensions, tightly-coupled peripherals or architectural modification. It requires however, configurable hardware in order to be deployed, but since it is architecturally non-intrusive it can be used on any COTS SoC with FPGA fabric.

The sniffer's architecture and interface is depicted in Figure 6.4. It is deployed in the FPGA fabric between the MSS and a memory block where the source architecture data memory shall be allocated. The instantiated memory block is part of the FPGA's RAM memory resources and has a simple and generic interface

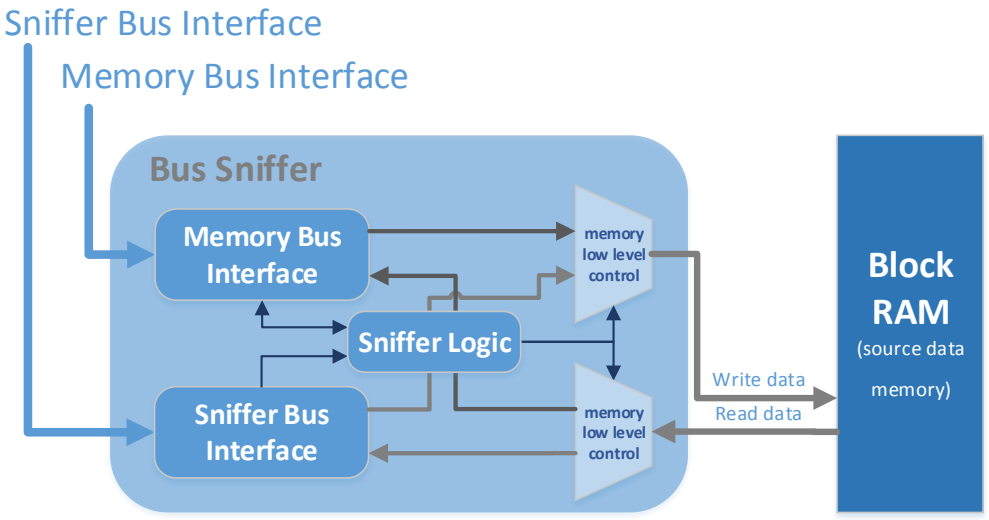


Figure 6.4: Non-intrusive hybrid acceleration architecture.

through address bus, read and write data lines plus write and read enable.

The sniffer has two MSS bus interfaces, one designated to the memory and another for sniffer configuration and operation. The bus interfaces used were the AMBA 3 AHB-Lite for both. Regarding the memory bus interface, it is known to the MSS as a regular and common memory access, and during DBT operation it behaves as one. The address line and the access signals are monitored through the sniffer logic, in order to identify accesses to sensitive memory addresses, and trigger an action if that is the case. Read and write access must be distinguished and handled accordingly: while a write access might just require an action to be triggered over the new value, a read access might require the execution of an action before the value being read. This requires that the read action must be interrupted or putted on hold before the modification take place and resumed after. The most efficient way to trigger the execution of those action in the MSS is through an exception or interrupt. Both possibilities are valid, however, a specific type of exception which particularly fits this application case, is the bus fault, because (1) it may be directly generated through the bus interface signals, (2) it is natively generated after a failed memory access and (3) upon its happening there are already mechanism which result in the re-execution of the failed instruction, which is particularly useful to the read access application of the sniffer.

Hence, an exception should be generated before a matching read access and after a write access, so that during a memory read any old values may be updated or copied to the target address, while during a memory write the actions to be performed act on top of the fresh data.

Table 6.2: Sniffer module FPGA resource utilization.

Comparators	4LUT	DFF	Combined Resources
1	242	164	406
8	746	393	1139
16	1322	654	1976
32	2474	1177	3651

The sniffer module must then be configurable with (1) the memory addresses of the desired location and (2) the trigger type of access. The type of access depends upon the application of the sniffer. In addition, there must be an adequate number of these access detectors working in parallel in order to support multiple functionalities, and a way to detect the exception source, i.e., which address has been accessed.

Finally, the sniffer module must have a secondary access to the data memory, to allow accesses to the sniffed memory locations during the exception, preventing a hard fault to happen. This was solved through a back-door memory access integrated in the sniffer bus interface. By accessing a higher addressing range of the sniffer address space, the access is diverted to the memory block RAM, bypassing the sniffer detection. The sniffer module was described in Verilog HDL and synthesized to the Microsemi SmartFusion2 SoC FPGA, resulting in the resource utilization presented in the Table 6.2 for 1, 8 16 and 32 address comparators.

6.3.2 Interface and Software API

The sniffer module interfaces the MSS through two AMBA3 AHB-Lite slave channels, one dedicated for the source data memory access and another for sniffer configuration and back-door memory access. The block RAM memory interface is fully transparent to the MSS. The integrations is performed by the linker who is informed of the address where the block RAM's AHB-Lite slave port is assigned at. The linker is also instructed to create a memory region spanning the block RAM and to place into that region a memory section containing the source data memory. This information is passed to the linker through the ILINK configuration file **.icf*, as displayed in Listing 6.2. The second interface is handled in an identical manner to the one presented for the Tcache hardware manager. There are interface registers mapped at specific memory addresses, which are presented at the Figure 6.5.

```
1  ...
2  /*-Block RAM Memory Region-*/
3
4  define symbol __ICFEDIT_region_RAM2_start__ = 0x30000000;
5  define symbol __ICFEDIT_region_RAM2_end__   = 0x30004FFF;
6
7  define region RAM2_region   = mem:[from __ICFEDIT_region_RAM2_start__
       to __ICFEDIT_region_RAM2_end__];
8
9  place in RAM2_region  { section SEC_RAM_8051};
10 ...
```

Listing 6.2: Linker configuration file (.icf) snippet.

The register at base address `0x51000000`, SNIFFER_CTR_REG, controls the general functionality of the sniffer module. The value `0x00000000` disables the sniffer while `0x00000001` enables its operation. Register SNIFFER_CMP_MATCH_REG holds the information of the access matches that occurred. Each of its bits is associated with the equivalent comparator. Upon an access match, the bit corresponding to the match comparator is set. The register SNIFFER_TYPE_MATCH_REG holds the information of the type of match that occurred on the

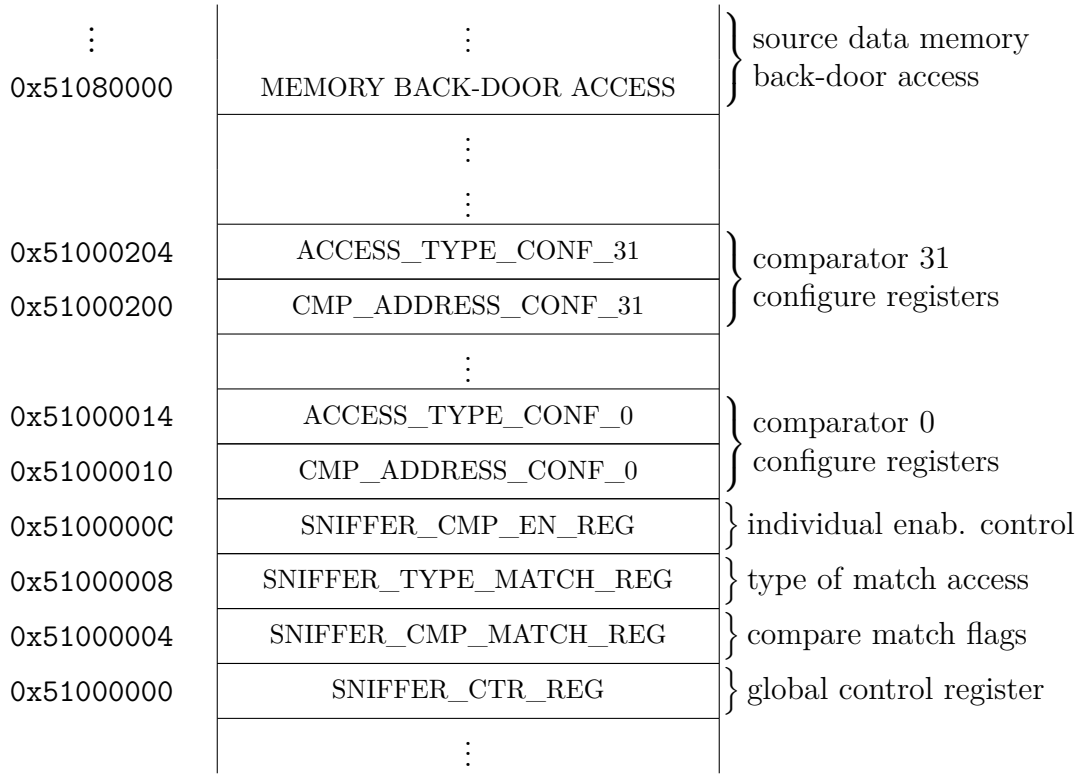


Figure 6.5: Tcache hardware manager peripheral mapping.

involved comparator. The value `1` signals a write access while `0` signals a read access. The register SNIFFER_CMP_EN_REG is used to enable or disable the comparators individually. To enable a comparator, the equivalent bit should be set, and vice-versa. The comparators configuration registers are the CMP_ADDRESS_CONF_n and ACCESS_TYPE_CONF_n. The address to be sniffed should be written on the former register. The type of accesses that generates a match is configured through the latter register: `0x0001` for read accesses, `0x0010` for write accesses and `0x0011` for both.

A simple API, similar to the one presented for the hardware Tcache manager, was also created. On a comparator match event a bus access fault exception is generated by the sniffer in order to call a handled and execute the required actions. On the target processor used, that exception is the *bus fault* exception, more specifically a *data abort* which natively occurs when an error response is received on a transfer in the AHB interfaces during a data read/write.

The procedure followed to handle an address match event is defined a follows:

1. Read the SNIFFER_CMP_MATCH_REG register to identify which comparator matched an address access. In the case that none of the bits is set, the exception was not caused by the sniffer module and should be handler accordingly.

2. Read the SNIFFER_TYPE_MATCH_REG register to determine which type of access (read or write) caused the match. If the comparator is known to have a read only or write only configuration, this step might be skipped.

3. Perform the necessary actions by calling the appropriate helper functions to handle the exception.

4. Clear address match bit on register SNIFFER_CMP_MATCH_REG.

The sniffer module's functionalities will be demonstrated in the following sections, trough its application in different case scenarios: to handle condition codes, remap peripherals and provide interrupt support to the DBT engine.

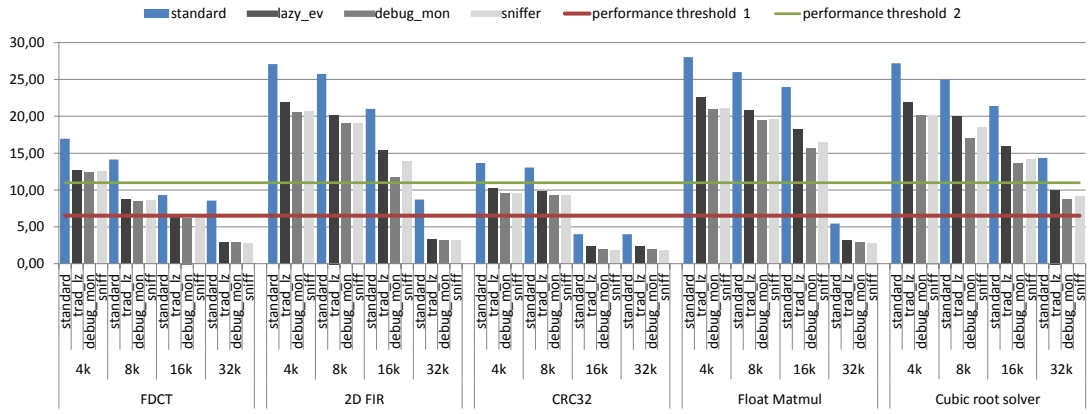
6.4 CC Handling

CC handling is a feature that requires great effort to be correctly emulated, as demonstrated in the previous chapter. Following the debug monitor-based approach and in order to surpass the drawbacks found over its reutilization, the non-intrusive hybrid acceleration architecture is proposed to be used in replacement of the debug monitor module. Through the sniffer module it is possible to trigger and evaluate the condition codes of the source architecture without the limitations of (1) losing the debug support from the debug monitor module and (2) the interrupt return stack manipulation. The debug support recovery is *per se* a determinant reason to use this approach, but in addition the stack manipulation might even reduce some cycles, with potential impact on the global performance of the system.

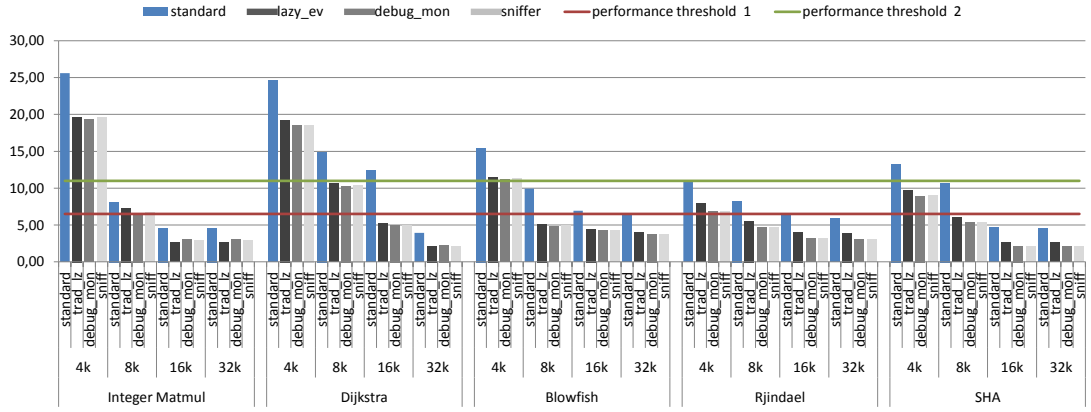
Hence the sniffer was configured to monitor the PSW special function address of the MCS-51 core for read accesses, so that the condition codes might be updated accordingly with the last affecting operation and operands, stored at the `CC_Parameter` structure. Any read access will cause a *data abort* interrupt, which will be dealt with, as described above and calling the `helper_CC()` helper function. For these tests the full legacy support approach was evaluated, using the hardware managed version of the Tcache. The same benchmarks were executed and the results are shown in the Graphics 6.6a and 6.6b in Figure 6.6.

The results show similar performance to the debug monitor based approach. In some cases (2D FIR 16Kb, Cubic root solver 8Kb) there is even some performance penalty regarding the debug monitor approach, however the sniffer based performs consistently better than the two other static approaches. From the performed tests, for every test and every Tcache size, the sniffer approach performance is on average 38% better than the standard method, 6% better than the lazy evaluation method and 0,2% worst than the debug monitor method. The expected gain from the lack of stack manipulation did not show in the test because they were overtaken by penalty induced with the instructions for enabling and disabling the peripheral between CC updates.

Considering the sniffer and debug monitor based tests results, performance-wise, there is no benefit from using the bus sniffer mechanism to trigger the CC update to the detriment of the debug monitor approach. Integration-wise however, it allows to take back the debug monitor functionalities, with similar results.



(a) FDCT, 2D FIR, CRC32, Float Matmul and Cubic root solver results.



(b) Integer Matmul, Dijkstra, Blowfish, Rjindael and SHA results.

Figure 6.6: Target/source global execution ratio, for the standard, lazy evaluation, debug monitor and sniffer based CC evaluation, using the hardware managed Tcache, for different Tcache sizes.

The resemblance in the performance tests results from the similarity of the methods. Both rely on an external trigger source to an interrupt/exception and consequent CC evaluation function call. Both methods also use the `CC-Parameter` structure (similarly to the other two methods) to save the operands and operations for CC emulation.

The structure manipulation during execution is heavy and time consuming, however can not be avoided with the evaluated methods. One option to avoid its use is the full emulation of the CC affecting instructions through a helper function during execution However such option may have other negative impacts on performance, and CC emulation would still be necessary. If reconfigurable hardware is available, a full source ALU replication as an acceleration module is suggested to be evaluated, using the sniffer module to trigger the PSW copy from the accelerator to its source memory location. This option, despite being more 'brute-force'

driven provides instant CC update and avoids the costs of emulation. Due to time limitations this approach was not attempted, and thus it is suggested as future work.

6.5 Peripheral Remapping

The sniffer module was also tested for functionality expansion and peripheral support, which is of great importance when dealing with embedded systems. Peripherals access is a feature with a great dynamic behavior. This dynamic affects its support on DBT because despite accesses can be predicted during translation, the performance penalty of such detections ripples across the whole system (requires verifying the addresses of every `gen_ld()`, `gen_st()` and related micro Ops). The implemented non-intrusive mechanism deals with it with ease as demonstrated in this section.

Among the multiple peripherals included in the MCS-51 architecture (e.g., timers, GPIO, Universal Asynchronous Receiver/Transmitter (UART)), the UART serial port communication device was selected. The functional verification of this peripheral is easy to attain, and it is perhaps the mostly used data transfer mechanism on legacy MCS-51 code.

In order to remap source into target peripherals, the supported peripherals of the source hardware must be known, as well as its configuration options, control and status registers and command bits. It is also necessary, although not mandatory, to find similar peripherals on the target architecture device. If no replacement peripherals are available the reconfigurable FPGA fabric might be used to deploy an IP module of the peripheral to be remapped. It may also, if decided in the project, to remap the peripheral into another type of peripheral, as long as the interfaces, type of data and type of transfer are possible to mutate between the original and the destination peripheral. Taking the example the UART port, it might be remapped and assigned for instance to a SPI device, as long as the original behavior of the source code remains intact. In this application case, the source UART port was mapped to a UART port available in the target Cortex-M3 platform.

On what concerns the MCS-51 UART configuration, there is one control register, the `SCON(0x98)`, plus the transfer buffer `SBUF(0x99)`. There are thee addi-

tional registers involved, for baud rate generation purposes, the RCAP2H(0xCB), RCAP2L(0xCA) and the T2CON(0xC8), which refer to the reload and capture timer 2 high and low value and the timer 2 control register, respectively. Every of these five registers must be added and monitored by the sniffer, thus a comparator must be assigned to each of them. A write access monitoring generated the type of exception that allows to know which configuration is being set for each of the registers. Only the SBUF requires a read and write access monitoring, because it is a shadowed register for both send and receive transfers. The exception triggered by any write to these registers will provide the opportunity to sniff the applied configuration by the source binaries, and replicate it in the assigned target UART.

For the purpose of testing the UART peripheral remapping, a simple UART polling transfer program was developed and tested on a native MCS-51 core (ATMEL 89C51IC2) with the message:

```
"Hello!  is it me you're looking for?!\n-MCS-51 - by polling\n"
```

The binaries were then set to translation and peripheral remapping. The described remapping configurations were added in the envReset() virtual method of the derived Translator class, which is also used to perform the necessary initializations. The function on the Listing 6.3 is called when the SBUF register is written with the byte to send.

The intercepted byte is then forwarded to the Cortex-M3 UART through the vendor provided API for a polled transmission. After completed the transmission, the TI flag of the SCON, which signals a transmit complete, is set and the exception returns to the execution of the translated code.

```
1  void CTranslator8051::helper_SBUF_WRITE(void){
2
3      uint8_t tmp;
4
5      tmp = BACKDOOR_GET_UINT8_T(env.dataMem[SBUF]);
6      MSS_UART_polled_tx(&g_mss_uart1, (const uint8_t *)&tmp, 1);
7
8      env.dataMem[SCON] |= 0x02; //set the TI flag to 1;
9      return;
10 }
```

Listing 6.3: SBUF write action helper function.

The translated code will eventually read and verify the TI flag in a loop (polled

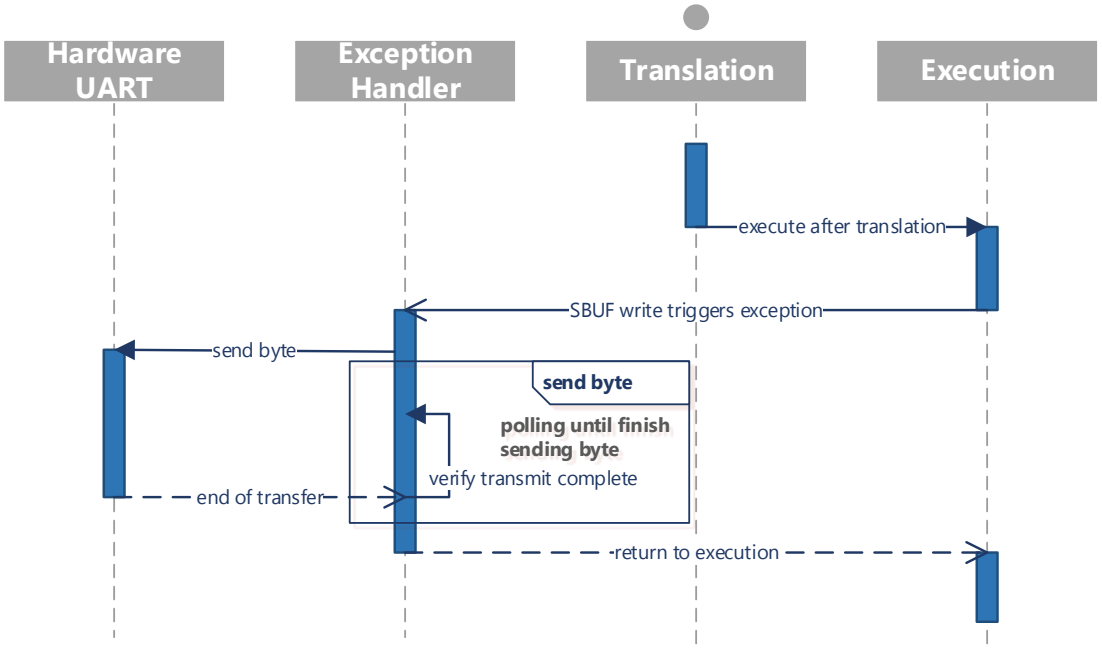


Figure 6.7: Peripheral emulation sequence diagram.

transmission) and since the byte was sent already it will pass the condition and proceed with the program execution to send the next byte. The described sequence of events is presented in Figure 6.7, where the code translation and execution domains are distinguished, together with the exception handler routine. The hardware UART device was also added to the diagram to express its role between the interactions as well as its parallel execution. The circle on top of the "Translation" domain expresses its role similar to an "Actor" since it is the principal process in the sequence due to the DBT engine being implicitly represented on it. The source binaries were loaded and translated on the DBT system. The target UART port was connected to a computer and serial port terminal tool was used to receive and

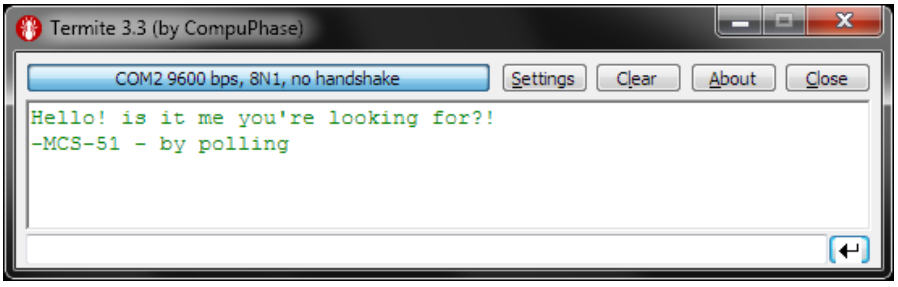


Figure 6.8: Received message from the MCS-51 remapped UART transmission by polling.

display the message. The received message and port settings (i.e., baud rate, data bits, parity bit and stop bit) are shown in Figure 6.8. The configuration of the serial port was correctly deciphered and applied to the target UART port, and the peripheral remapping mechanism proved to be functional.

6.6 Interrupt Support

The presented functionality of the sniffer module is explored in this section for interrupt support. For exemplification purposes, and similarly to the previous section, a type of interrupt present in the source architecture was selected to be emulated. The UART interrupt was elected for the demonstration because it also produces a visible output that eases functionality confirmation and because it builds up on top of the demonstrated peripheral, as well as demonstrating the scalability of the non-intrusiveness architecture, spanning the peripheral supported modes and creating a case which intercalates the peripheral with the interrupt support. A simple program for sending bytes through the serial port based on interrupt was written and compiled for the MCS-51. The message to send was the following:

```
"Hello!  is it me you're looking for?!\n-MCS-51 - by interrupt\n"
```

On what concerns the serial port communication, the configuration registers monitoring and mapping to the target UART remains the same, but with the extra step to write to the register `IEN`, Interrupt Enable control, required to configure the interrupt mechanism to be used together with the serial port.

Generally the interrupt handling mechanism is based on a halt in the data-path, after completion of the already fetched instructions, and a jump to a code location where lies the ISR. There are context saving operations that must take place in order to not affect the normal program flow data, since this is an unpredictable event not involving a *caller* that usually saves the relevant process data. Upon processing of the ISR, the context data is restore and the processor resumes the execution of the code. To emulate this mechanism into DBT there are some considerations to take into account. On what concerns the interrupt cause and processing:

1. Similarly to the peripherals, the interrupt source must be also remapped to a target interrupt, so that no polling is involved in the event registration and

the source translated code has available processing resources to perform its eventual computations.

2. The source ISR code locations must be known, in order to be located and translated.

3. A mechanism to interrupt the execution of translated code and give place to the execution the ISR must be provided.

4. A mechanism to interrupt the normal DBT cycles of code Translation and Execution without disrupting it must also be provided, so that the source ISR may be translated and executed in the middle of the DBT flow and posteriorly return it.

On what regards the context switch the considerations are:

5. The integrity of the source translated code under execution must be assured, in order to maintain code correctness. Contrary to the native execution where there is a concept of atomicity associated with each instruction, on a one-to-many type of translator that concept is hard to replicate.

6. The interrupt emulation process must also not interfere with the architectural resources used in the DBT process, namely, the intermediary work registers used during code execution, the execution stack and the return address.

These considerations were taken into account and the proposed interrupt emulation mechanism addresses is conformable with them. The target UART peripheral is configured for an interrupt based operation, which allows to (1) use the target interrupt source to trigger the source interrupt handling action. On the MCS-51, (2) the ISR code locations are determined by the architecture and for the serial port interrupt, it is the address `0x23`. Hence after a UART remapping, (3) if an interrupt of the target UART occurs, the code execution is halted and the source ISR code to be translated and executed from, is known to be at the source address `0x23`.

In order to not break the ordinary DBT flow, a supplementary DBT process is created for the purpose of translating and setting the execution of ISR code. It is implemented as a method `trigger_interrupt(uint16_t isr_addr)` in the base Translator class, and received the ISR location address as parameter. When a (4) remapped interrupt occurs, this method is called and saves the current `env.PC` (which controls the DBT flow), replaces it with the ISR address parameter and

starts its translation as regular BBs. The translate and execute cycle is also replicated in this method, until the translation and execution of the source instruction `RETI`, which determines the return from an interrupt. At this point the original `env.PC` is restored and the normal execution proceeds after returning from the target interrupt.

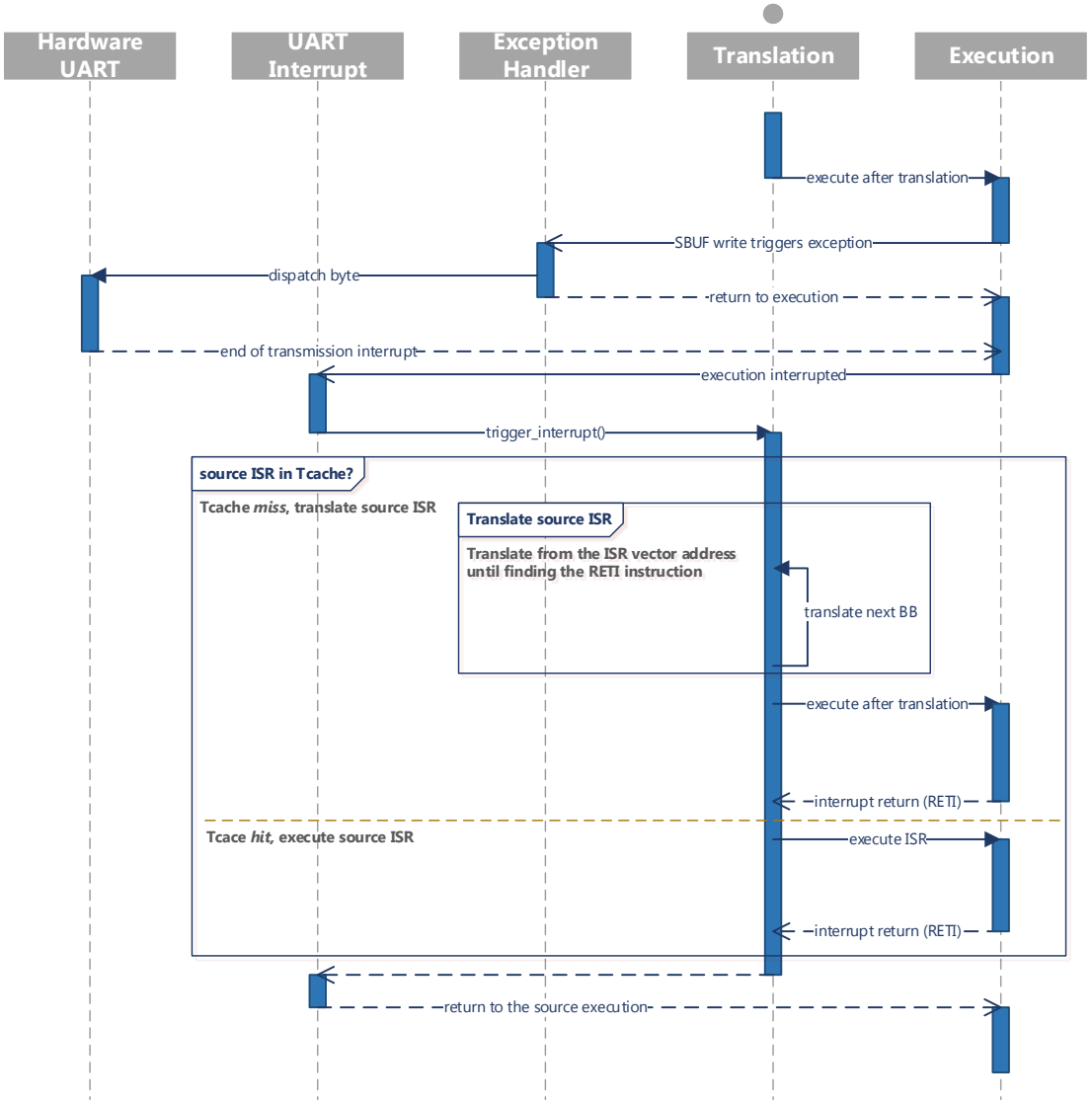


Figure 6.9: Interrupt emulation sequence diagram.

Regarding the context switching integrity, (5) the MCS-51 ISR proceeds to save and restore any register used, and on its turn the Cortex-M3 finished all the instructions in the pipeline prior to attend interrupts. This may originate that all the target instructions that form a source single instruction are interrupted during execution. The concept of source instruction integrity is not respected if that happens, however no consequences arise from it due to the atomicity being

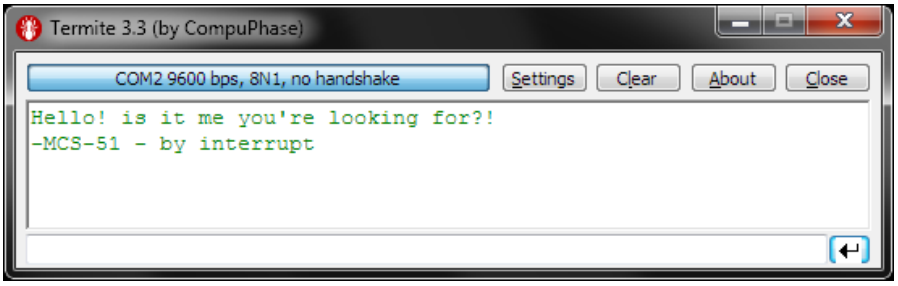


Figure 6.10: Received message from the MCS-51 remapped UART transmission by interrupt.

neatly broken at the target instructions. The last concern is addressed by (6) the target processor context saving during interrupt attendance and by the choice of preserved registers for working registers. After an interrupt event the ARMv7 architecture pushes the context registers (and also the scratch registers) into the stack, restoring them during interrupt return.

Figure 6.9 illustrates the interrupt support sequence diagram. This diagram, in addition to the one presented in Section 6.5, has the "UART Interrupt" domain, which represents the target ISR handler routine. The initial part of the diagram, which depends on the sniffer module, is similar to the previous diagram, but now the exception handler initiates the transfer and does not waits until its end. Instead, the control is returned to the Execution of the source binaries. From there on, the proposed ISR handling mechanism takes place until the end of the execution of the source ISR handler. After returning from the target UART Interrupt the sequence is repeated, but in case the source ISR is already at Tcache, then no code translation is performed and it is directly executed. If, the `SBUF` is written during the execution of the source ISR, the peripheral mapping mechanism is triggered, the interrupt handling halted and a new byte is sent to transfer, and then the ISR is resumed. The results of this implementation are shown in Figure 6.10, where the new message is shown in the serial port terminal tool, after being correctly received.

6.7 Conclusions

In this chapter the functionality extension of the DBT engine using external hardware support was addressed. A Tcache partially deployed in hardware and a COTS compliant architecture for DBT functionality extension and hardware acceleration

were presented.

The Tcache hardware management with a circular BB registry dismisses the use of a hash table or other secondary software management mechanisms, resulting in significant performance enhancement, compared to the previously presented hardware-only management methods, for every test scenario. The hardware look-up-table has a reduced and fixed insertion and search time, leading to better performance and scalability. The results are on average 25% and 26% better than the linked list and hash table implementations, respectively

The proposed acceleration and functionality extension architecture is based on an external hardware module, integrated as a bus sniffer, which results in a hardware and non-intrusive execution flow technique, never tried before in the state of the art. The bus sniffer may be used to trigger software or hardware components, based on the source architecture memory accesses, providing great flexibility on the type of application to serve without disturbing the base DBT program flow. Three types of application to the proposed bus sniffer were presented: to handle the CC, to remap source peripherals and to provide interrupt support. On all the use cases the bus sniffer approach functionality was tested with a practical demonstrator. The efficiency of the approach may only be evaluated in the CC handling case due to lack of comparative results for the peripheral remapping and interrupt support cases. However, the type of cases used, which have exact timings and sequences (e.g., baud rate, sending order), posing an increased integration challenge, prove the validity of the approach on what concerns functionality correctness. An in-depth discussion on each of the application cases of the bus sniffer now follows.

Regarding the presented bus sniffer based CC handling, the results are similar to the debug monitor based approach. For a full legacy support execution, the performance is superior to the two other software-only approaches. Using this method for CC evaluation allows the use of the available debug resources while reducing the CC handling time.

For the peripheral remapping, the presented approach attained a mechanism to emulate and remap peripherals in DBT. The mechanism is transversal for other communication peripherals and flexible enough to extend the supported functionalities beyond the ones demonstrated in this application case. On what concerns timers, its direct applicability is not so clear. The configuration sniffing and remapping of the source timers to the target timers is supported (as long as the code to map the equivalent configurations is added to the translator), however and due to

differences of clock domains, performance penalties and general de-synchronization between the native and translated execution, this topic requires further efforts in order to be presented.

Regarding interrupt support, we propose a novel mechanism to handle interrupts in DBT, by using target interrupts trigger to execute source ISR, in a fully dynamic manner. Interrupt latency was not addressed because this study is far from real-time requirement systems.

Considering the increasing variability and configuration complexity added to the already complex DBT system with the presented functionalities, it is now mandatory to promote, along with the DBTor, dedicated tools for managing the complexity and easing the final systems customization, toward a unified DBT framework. This goal will be address in the next chapter.

Chapter 7

Automation Enablement through DSL

Embedded systems design for industrial application solutions is a highly complex topic due to the challenge of integrating multiple technologies into a single solution, the inherent complexity of the problems to be solved and also because the proposed solutions often require a great level of interoperability among their components and also the outside world. DBT has been used as a tool to deal with such interoperability issues, e.g., legacy support, virtualization and secure execution, among others, as already presented in Chapter 1. However its integration in the industry as an end-product is hampered by the required intricate variability management. To address these issues and in an attempt to empower DBT utilization through an interoperability-providing tool, it is proposed a model-driven DSL modeling language for DBT architectures. The developed DSL proved to be efficient to model the in-house DBT engine, and FAT-DBT, a framework for ready-to-use DBT solutions was obtained. FAT-DBT provides design validation, easy configuration of customizable DBT parameters and components, as well as code generation features.

7.1 Introduction

Some of the big difficulties in todays' industrial solutions design is the high complexity of the systems and the variability management challenge [136] that the integration of multiple technologies poses. This is partially caused by the inherent

complexity of the problems that need to be solved but also because the solutions often require a great level of interoperability between the system's elements and the outside world. A long known solution to deal with such interoperability issues is BT. In this IoT advent, DBT may become an emerging technology due to the interoperability it may provide to heterogeneous industrial environments, and also for its usability as a tool to support competitors products' firmware on a market race perspective [45,137]. In spite of being successfully deployed in general-purpose systems to support cross-ISA binary compatibility, dynamic optimization, profiling, virtualization, secure execution or debugging environment, in embedded systems, binary translation has been avoided mainly due to performance, memory and power overheads [45, 61, 94].

However, DBT utilization as an end-product in the industry has been hampered by the complexity of the subject and its associated variability management, which brings configuration challenges into the final solution [94]. The accessible and profitable use of DBT requires design automation paradigms and variability management solutions, expanding its usage for DBT laymen. In this sense, a DBT framework must provide support for several source and target architectures, optional execution features (e.g., code profiling, type of Tcache management, peripheral remapping, etc.), resource utilization settings (e.g., available memory size, Tcache size), design validation and consequent automatic code generation.

In this chapter a model-driven DSL modeling language for DBT architectures is presented, aiming to improve complexity management, design validation and industry interoperability. Along with the DSL modeling language a framework named FAT-DBT was also developed, for ready-to-use DBT solutions, providing easy configuration and code generation features, even for DBT laymen users. The remaining of this chapter is organized as follows: Section 7.2 introduces the DSL subject theme; Section 7.3 presents the modeling Elaboration Language (EL) and the FAT-DBT framework; the DBT modeling process with the EL is explained in Section 7.4; Section 7.5 describes the implementation of the DBT system using the obtained framework and in Section 7.6 and 7.7 the work evaluation is presented and the conclusions are exposed, respectively.

7.2 DSL for DBT

A DSL is a programming language that targets a specific problem domain. A DSL should not provide features to solve every kinds of problems found in a certain domain, but instead should make it easier to deal with the problems of the domain it is specific for [138]. The usage of a DSL over a GPL is justified by several advantages such as (1) gains on expressiveness on the target domain, (2) ease of use, (3) enhanced productivity, (4) reliability, (5) maintainability, (6) easier reasoning and validation, and (7) the direct involvement of domain experts [139]. The effort required to develop a DSL is however quite hard, as it requires a lot of technical experience and great understanding of the domain. Nonetheless, after implemented, the development work usually pays off [140]. The interest in DSLs for GP [141] and in MDD [142] is becoming wider, as they promote software reuse and fast development through a high abstraction level. GP purpose is to automatically generate a system given a set of specifications [141]. MDD is an approach used to create extensive and descriptive system models on a higher implementation abstraction, thus simplifying development and testing activities [142]. Together, these two techniques promote software reutilization and automatic code generation, powering the DSL to map different models together and elaborate the final system code [143].

To pursue such goal, a model-driven DSL, named EL, was developed to automatically generate code from the source files of a given system's model. It is based in the Service Component Architecture (SCA) standard, which specifies that a model's components should follow a composite pattern [144]. SCA features six key elements: (1) composite, (2) component, (3) service, (4) reference, (5) property and (6) wire. A complete reference architecture can be constructed by identifying system components and their interactions, as well as the properties associated to each component. The EL grammar was developed using Xtext [145], which is a framework for programming languages and domain-specific languages development, offering a full development infrastructure including parser, linker, type-checker and compiler [146]. An auxiliary tool, Xtend [147], was also used to implement the language validators, the code generation software and other additional features. Xtend is a flexible and expressive dialect of Java, which compiles into readable Java 5 compatible source code [147]. Both Xtend and Xtext are widely used for DSL development and integrable with Eclipse IDE [138]. The EL development will not be covered in this thesis. The EL was used as a solution

for the modeling of a DBTor for embedded systems. The obtained tool, together with GUI component configurators, was called Framework for Application Tailored Dynamic Binary Translation - FAT-DBT.

7.3 FAT-DBT - EL and Framework Overview

The EL framework work-flow, shown in Figure 7.1, is composed by four main stages: (1) modeling, (2) elaboration, (3) configuration and (4) code generation. During each stage a set of artifacts is created and then used in the following stages.

The modeling stage goal is to create the model that will be used as the system's reference architecture. In this stage every component must be identified, as well as their dependencies, properties, interfaces and relations with other components. This model must be described through the EL. Then, the model representation goes through the model compiler, which proceeds to syntactic and semantic vali-

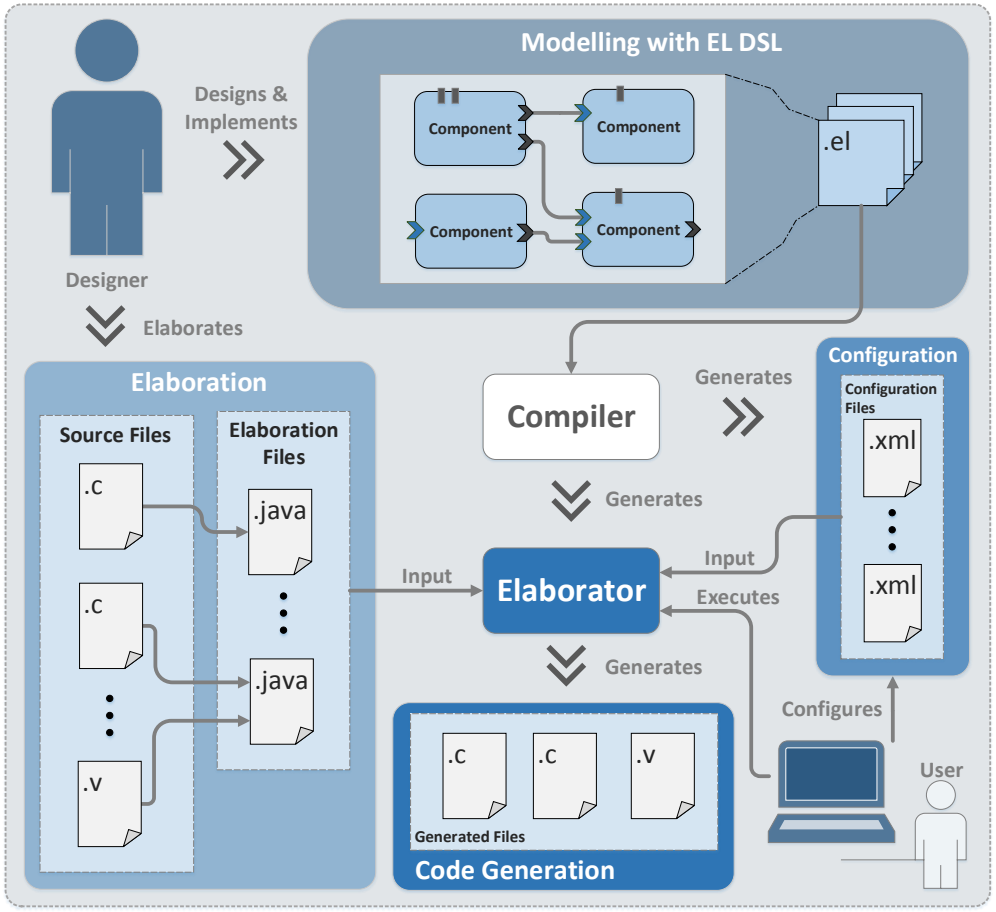


Figure 7.1: Elaboration Language framework workflow.

dations. After a successful compilation, an architecture-specific Java Elaborator is generated, together with a set of XML configuration files, and abstract elaboration and Java classes for each component.

In the elaboration stage, the elaboration files gather the information from the whole component's behavior and how the source code for that component must be generated. If there is more than one implementation available to a given component, the desired implementation should be specified in the correspondent configuration file. Only one elaboration class per component is executed by the Elaborator. The annotation process is eased by an API that was created to fetch the desired values from the configuration files and replace them within the source code files. The elaboration stage also includes the definition of the annotated source files and the implementation of the elaboration classes (which are based on the abstract elaboration classes).

During configuration stage, it is possible to modify the component properties' values and which elaboration file will be loaded into the Elaborator. It is also possible to have specific properties for each elaboration which are not presented in the reference architecture. In this case another XML file must be provided by the developer.

Finally, the generated elaborator is executed. During this process, components' properties are fetched and the elaboration classes are loaded using Java reflexion.

EL's Constructs

As previously stated, the EL follows a SCA, which means that an EL file contains the following constructs: interfaces, languages and components. An interface is a set of functions that implement a service provided by a component. Components can be connected through bindings of services and references that follow the same interface type. Every declared component must have an implementation language. The language used by these components should have, in addition to a name, an annotation section where the user defines a set of characters that will be used to define annotations in the source files. A component can be composed by a set of subcomponents, properties, references, services and a free section used to make assignments or promoting services and/or references. It can also enclose other components and access all of their properties, references and services. A component can also inherit another component. This operation transfers all the

Table 7.1: Available EL's keywords.

Keyword	***Description***
`annotation`	Defines the character that limits the annotations.
`as`	Renames a promoted reference or service.
`bind`	Binds a reference to a service.
`compile`	Tells to compiler which is the top level component.
`component`	Defines a component.
`final`	Defines that a component has a concrete elaboration.
`import`	Imports the content of the specified file.
`interface`	Defines a set of functions used by a service or pointed by a reference.
`is`	Inherits the specified component.
`language`	Defines a language.
`promote`	Promotes a reference or service from a subcomponent to a component.
`properties`	Defines the properties set of a component.
`reference`	Defines the reference used in a promote or in a bind operation.
`references`	Defines the references set of a component.
`restrict`	Restricts the values that a property can take to a user's defined set.
`service`	Defines the service used in a promote or in a bind operation.
`services`	Defines the services set of a component.
`to`	Connects a reference to a service in a bind operation.

content of the inherited component to the top-level component. Each property may have its own restriction list or range, for value filtering purposes. Later, the specified properties will receive their values in the assignments section or at the code generation step, through the replacement of the established annotations. By the definition of a composite design pattern, a component can provide one or more services to other components. Since a service is implemented by interface functions, a component can not have more than one service on the same interface. A reference should be created only when a component requires to access a subcomponent's service. The `assignments` field allows the user to set the value of components and subcomponents properties. In the `promotes` section the user can promote references or services of a subcomponent in order to use them in their top-level component. Furthermore, it is also possible to bind references to services in the `binds` section and to specify the top-level component on the hierarchy using the keyword `compile`. This specification also indicates the classes' invocation order to the Elaborator program. Table 7.1 presents the available EL's keywords and their respective description.

7.4 Modeling the DBT Engine

The DBT system's model building was supported on the theoretical and implementation background on dynamic binary translation, obtained from the in-house DBT engine development. After a thoughtful analysis of the existent deployment, several components and interfaces were identified. Moreover, several configuration points were found and transposed to the model through properties. The end goal of the model was to automate the system configuration, to perform architecture validation and to generate the final source code for the end system.

Reference Architecture

In Figure 7.2 it is represented the reference architecture model for the DBTor with a simplified representation of the most relevant interfaces. The model components are represented as blocks and the properties as black diamonds. The services are identified by gray polygons and references by blue polygons. Dashed lines are used to represent interfaces between components. The composite DBTor is made by all the other components and composites of the DBT reference architecture.

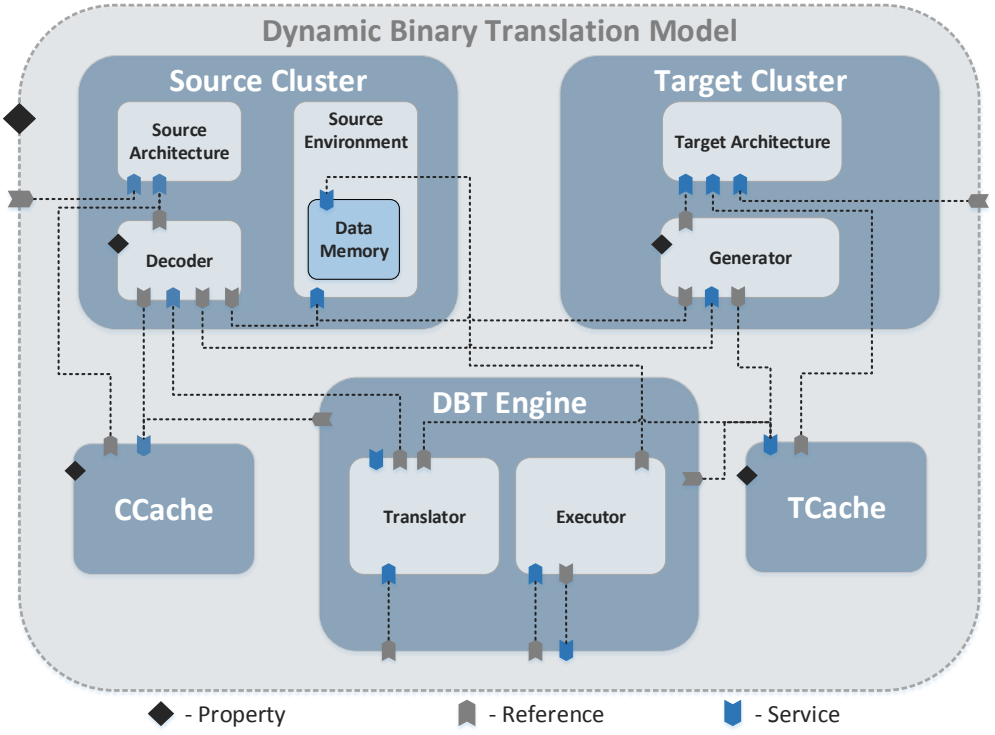


Figure 7.2: Reference architecture.

Through its modeling, five main sub-components were identified, all providing the main features of the system: Ccache, Tcache, Source Cluster, Target Cluster and DBT Engine. The Source Cluster is the composite that aggregates the software blocks associated with the source architecture, which are Source Environment (also composed by the data memory), Source Architecture and Decode. The Target Cluster is also a cluster of related software blocks, but in this case, related to the target architecture. It is composed by the Generator and Target Architecture components. The DBT Engine represents the heart of the translator. It models the intermediary layer created at run-time that implements the translation of source code and execution of the target code. Therefore, it is composed by two sub-components: Translator and Executor.

7.5 Implementation

Based on the DBT reference architecture, an EL code representation was created for all the components and interfaces. Listing 7.1 depicts the code representation of the `TranslationCache` component with its properties, references and services. The component properties come with default values that can be later modified by the end user. The `TranslationCache` has a reference, `r_ISA`, to the target architecture to define the Translation Cache word size and the service `s_TCache` provided by the Translation Cache. This service interface is represented in the Listing 7.2, with its five functions: `addTag`, `getTransAddr`, `getLastTransAddr`, `getCurrInsAddr` and `cacheCode`.

```
1  component TranslationCache(cpp)
2  {
3      properties:
4          int TCache_Size : 8192  // 20Kbytes
5          string cacheType : "Flush"
6      services:
7          i_TCache s_TCache
8      references:
9          i_ISA r_ISA
10 }
```

Listing 7.1: EL representation of TranslationCache component.

```
interface i_TCache
{
    addTag
    getTransAddr
    getLastTransAddr
    getCurrInsAddr
    cacheCode
}
```

Listing 7.2: EL representation of the i_TCache interface.

The implementation of this component is written in `C++` language, therefore a language type entity `cpp` was created, where the meta-characters used to annotate the source files are defined as `'@@'` (Listing 7.3).

```
language cpp
{
    annotation: "@@"
}
```

Listing 7.3: EL representation of C++ language.

These meta-characters are used in the source code preceding and proceeding the unique identifier of an annotation that will be replaced by its respective value during the elaboration process of the final sources.

Elaboration

After the EL files being compiled, the elaboration files in Java and configuration files in XML are generated. In this phase the annotated source code files and configuration files are used in the elaboration process of the final files. The types of files used in the elaboration stage are described ahead:

- **Configuration Files** - The configuration files provide to the user the capability of modifying the default values of the component's properties and the system's behavior through the modification of the default elaboration files.
- **Annotated Sources** - While building the model, the designer identifies several configuration points that should be annotated in the source code with the meta-characters defined in the EL. The Listing 7.4 shows the annotations `TCacheSize`, `cacheType` and `CC_type`, that will be replaced by user

defined values during the elaboration process. These values will replace the (`@@TransCache_Size@@`, `@@TransCache_Type@@`) and `@@CC_type@@` annotations, respectively. `TCacheSize` and `cacheType` refer to the Tcache available size to store translation and the type of cache management used, respectively, while the `CC_type` expresses the method used in the DBT to emulate the source condition codes.

- **Elaboration Files** - The specific elaboration files for each source file has the annotations that should be replaced and their corresponding value. The elaboration API provides methods to replace the annotations and generate the final files. The annotations that were previous explained are presented in the Listing 7.5 with the replacement methods.

```
1 #define TCacheSize @@TransCache_Size@@
2 #define cacheType @@TransCache_Type@@
3 #define CC_type @@CC_type@@
```

Listing 7.4: Example of annotations.

```
1 openAnnotatedSource("defines.h");
2 replaceAnotation("TransCache_TSize",
     target.get_TCache_Size());
3 replaceAnotation("TransCache_Type", target.get_cacheType());
4 replaceAnotation("CC_type", target.get_CC_type());
```

Listing 7.5: Elaboration file.

In the header file it is necessary to change the annotation `TransCache_Size`, `TransCache_Type` and `CC_type`. These annotations should be replaced by the value of their respective properties, previously configure by the user. The elaboration file also contains methods that return the names of the implemented services specific for each elaboration.

7.6 Evaluation

In order to demonstrate the contribution of the EL, a reference DBT architecture framework was created, the FAT-DBT. After the system's configuration, all source files are automatically generated and ready to be compiled. The reference model abstracts the component's implementation to the user but requires the system de-

signer to specify the real implementation. An XML configuration file for the Translation Cache component is created, where `SpecificTranslationCacheElaborator` is specified as its implementation. The translation cache XML configuration file is shown in Listing 7.6. Every property requires a value as input, which can be specified by the end user or retrieved by its default value. Without these, it is not possible to generate the final DBT source code.

```
<component type="TranslationCache">
  <elaboration default="SpecificTranslationCacheElaborator">
    SpecificTranslationCacheElaborator
  </elaboration>
  <properties>
    <property type="int" name="TCache_Size" default="8192">
      <value>
        <element>4096</element>
      </value>
    </property>
    <property type="string" name="cacheType" default="Flush">
      <value>
        <element></element>
      </value>
    </property>
  </properties>
</component>
```

Listing 7.6: TranslationCache specific XML configuration file.

The translation cache default size is 8192, but it was configured as 4096, so the second value will be used in the compilable source code. The cache type is left to default (full flush eviction). The generator component is also configured with the condition codes evaluation type, so the respective XML configuration file was created. `SpecificGeneratorElaborator` is specified as the generator component's implementation. The generator XML configuration file is shown in Listing 7.7. The method for conditions codes emulation was modified from *lazy* to *standard evaluation*. The optimizations were also left to default and remained disable.

Figures 7.3 and 7.4 illustrate the graphic user interface (GUI) XML Component Editor interface that allows XML component configuration by the end-user. In these two examples, the components being configured are the TranslationCache (Tcache) and the Generator, respectively. For the TranslationCache component, two options are available for configuration in the Component Editor interface, the TCache_Size, and the cacheType, both displaying the default value extracted

form the XML configuration file, and accepting new configuration values, which will later replace the respective annotations during the code generation process.

```
<component type="Generator">
  <elaboration default="SpecificGeneratorElaborator">
SpecificGeneratorElaborator
  </elaboration>
  <properties>
    <property type="bool" name="optimizations" default="false"
      <value>
        <element></element>
      </value>
    </property>
    <property type="int" name="CC_type" default="1">
      <value>
        <element>STANDARD</element>
      </value>
    </property>
  </properties>
</component>
```

Listing 7.7: Specific XML configuration file.

For each property, the application restricts the inputs, according to their type (defined by the EL keyword type) or possible range (set in the field Value), avoiding wrong user inputs. The same type of interface is presented for the Generator component, in Figure 7.4, presenting the configuration possibility for the CC_-type (CC evaluation mechanism) and optimizations.

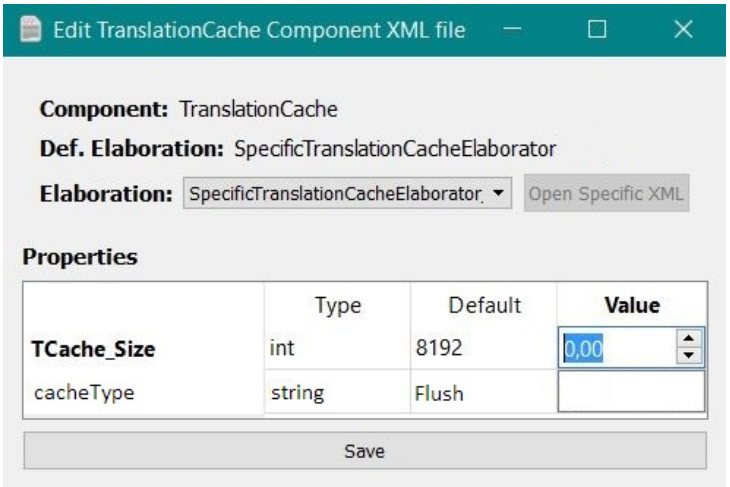


Figure 7.3: XML Component Editor interface for the TranslationCache component.

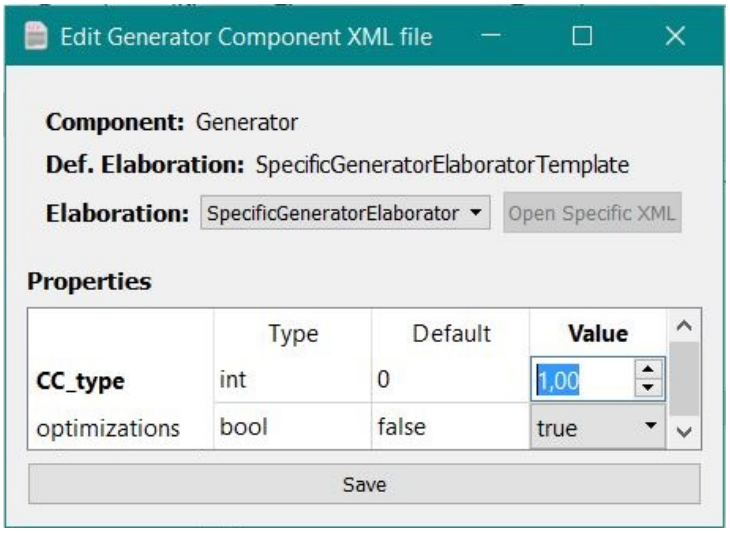


Figure 7.4: XML Component Editor interface for the Generator component.

After generation, the DBT source code was compiled and the result deployed in the development platform, and evaluated with the BEEBS benchmarks. There were no variations in the resulting code, neither in the program's performance. The output is shown in Figure 7.5, where the configuration inputs result can be seen. The translation cache size 4096 bytes (`0x1000`) and the `STANDARD EV` condition codes emulation settings are output to the console prior to the binaries execution.

```
MicroSemi SmartFusion2
  Device running on
        M2S150

Configuration settings:
Translation Cache size: 0x1000
Conditon codes handler in use: STANDARD EV.

Starting runDBT now...

FDCT benchmark about to start...
env.dataMem @ 0x30000000
env.dataMem[PSW] @ 0x30000150
Source program at 0x60040000 address, with maximum size of 4443 bytes
Exit address at 0x3
FDCT cycles spent 37367073


2D FIR benchmark about to start...
```

Figure 7.5: The compiled DBT source files executing on the evaluation board, translating a program written for MCS-51 architecture.

7.7 Conclusion

This chapter presented the FAT-DBT framework, the result of a modeling DSL for DBT architectures targeted for in embedded systems. The developed DSL, supported by the composite pattern, was used to modeled a DBTor, creating a higher abstraction level for system description. The attained functionality demonstrate that despite a DSL being hard to create, its potential contribution to the modeling, configuration and code generation of a certain domain tend to pay off, achieving higher productivity, lower development time and ease of use.

The proposed framework, FAT-DBT, aims to aid design automation, decreasing configuration efforts and promoting the use of DBT technology in the industry as a ready-to-use solution. The developed DBTor was modeled using the EL and integrated in the FAT-DBT, from where it is possible to be configure through the GUI XML Component Editor interface at two levels, (1) at data support components size (e.g., TCache size) and (2) at a functional level, through the customization of different functional mechanism, like the CC emulation method. The framework provides the configured translator's end source files without manual intervention on the DBTor's source code and without any code overhead.

Chapter 8

Conclusions and Future Work

This thesis proposed a specialized DBT engine for legacy support of resource-constrained embedded devices. The presented solution integrates customization features, resourceability and retargetability characteristics, for ease of configuration and porting. A hybrid DBT architecture solution was explored by resourcing available hardware and FPGA fabric for overhead mitigation and functionalities support. The final solution was integrated on a complete framework for ready-to-use DBT solutions.

This chapter concludes this thesis, discussing in Section 8.1 the answers to the previously posed research questions, and identifying in Section 8.2 the limitations found throughout the development of this work. Section 8.3 suggests future improvements and points possible research directions and finally, Section 8.4 lists the publications that resulted from the developed work.

8.1 Conclusions

Despite DBT appearance purpose being tightly connected with legacy support and binary compatibility, it became widely adopted in the computer arena due to its versatility and applicability in many other applications. The code instrumentation and optimization opportunities that derived from its use, opened new paths for software acceleration, system virtualization and simulation, among many other applications. However, the use of DBT has associated overheads that result from its deployment as an intermediary layer between the application binaries and

the execution machine. In order to benefit from the required DBT functionalities, while maintaining bearable amounts of overhead, software and hardware optimization techniques must be used. When considering resource-constrained embedded systems as the target machines for DBT, special attention should be dedicated not only to address the overheads, but also to the limited resources that these host systems provide. An effective deployment requires a specially tailored and resource-aware DBT engine, capable of delivering the BT functionalities, while maintaining the overhead at tolerable levels. The support of legacy binaries in modern resource-constrained embedded systems motivated the study of these issues, leading to the main research question raised in the Chapter 1 of this thesis:

How to leverage an optimized and accelerated dynamic binary translator, targeting resource-constrained embedded systems for legacy support?

The main question originated four different research directions, which were addressed through the remaining seven chapters.

To answer to the first question, **is DBT a possible solution to address the legacy support challenges on the resource-constrained embedded systems**, an assessment of popular legacy and cost-efficient modern embedded architectures was driven in Chapter 2. It lead to the selection of the Intel MCS-51 as a good legacy architecture example, while adopting the the ARMv7-M architecture as a deployment ISA, because of its reduced price, wide availability and popularity. During Chapter 3 and Chapter 4, an in-house DBT engine was developed from scratch because the state of the art failed in providing a cross-ISA DBT solution portable to embedded systems. The DBTor was designed, implemented and instantiated to the mentioned source and target architectures, tested and evaluated using the BEEBS benchmark suite. The results have shown that despite the existence of considerable overheads, it is possible to attain similar or even faster execution of legacy binaries in a legacy support DBTor running in a low-end processor architecture. The DBT execution was characterized, and different overhead sources were identified, with particular emphasis on the CC emulation process.

The second research question's intent was to identify **how to attain a flexible, yet application-tailorable solution, while minimizing NRE development costs and maximizing the solution applicability**. This question was also addressed in chapters 3 and 4, through the design of a architecturally agnostic DBTor with a resourceable and retargetable architecture. The use of an IR as a

source/target ISA abstraction layer and the deployment variability management through OO-techniques allowed to minimize NRE efforts for multiple ISA portings of the DBTor. Different Tcache management policies and sizes, as well as CC evaluation methods, were also easily integrated in the DBTor as customization options.

To address the question on **how to address the overheads associated with DBT, considering the low resources of the target devices, while providing full legacy support**, different software and hardware techniques were used to address multiple sources of overhead. In Chapter 5, the CC emulation overhead was tackled through software-based LE, and a novel technique using COTS hardware debug features. The innovative approach was evaluated against the LE and the standard evaluation, demonstrating its superiority on what concerns performance and functionality. Then, in Chapter 6, different hardware acceleration efforts were developed for overhead mitigation on Tcache management and CCs handling mechanisms. The hardware acceleration architecture's versatility was also demonstrated by providing support to peripherals and interrupts emulation. The hybrid Tcache with hardware management achieved superior performance compared to the software-only versions. On what concerns CC emulation, the hardware-based approach did not exceeded the debug monitor-based performance. However, it allowed to recover control over the debug functionalities of the target processor that become unaccessible when using the debug hardware for CC evaluation. The peripheral and interrupt emulation support functionalities were tested with two UART transmission binaries, by polling and by interrupt. The remapping of the peripheral was efficient enough to ensure accurate transmissions at a baud-rate of 9600 bits per second.

Regarding the last question, **how to manage system variability and enable solution space exploration and automation**, it was addressed in Chapter 7. To answer this question, a DSL for DBT architecture modeling was developed, creating a higher abstraction level for system description. Then, an embedded systems' DBT architecture was described and a framework called FAT-DBT was designed, providing DBT customization through a GUI for DBT design validation, system configuration and automatic code generation. FAT-DBT contributes to design automation, decreasing configuration efforts and promoting the use of DBT technology in the industry as a ready-to-use solution.

8.2 Limitations

Several limitations were identified during the development of this solution, either because were intentionally left outside the scope of this thesis, or because were found during its development. Such limitations are identified as:

- **Energy consumption analysis and power optimization** is a relevant topic that was not addressed due to the already complex nature of the proposed research topic and the expected workload. However, and specially considering the type of systems addressed, energy consumption values will depend on the host processor's workload. Nonetheless, the energy consumption is expected to decrease with the improvement of the DBTor.

- **Application level DBT** was not explored and is not supported. This type of translation would require an OS running in the host system, which was left out of the solution for resource-saving purposes. An additional cross-OS system-call conversion layer would also be necessary, introducing more overhead in the final system.

- **Multi-thread and/or multi-core DBT approach** is a limitation that would also require a host OS at the target system, and was not approached either. Despite the few task parallelism opportunities identified in the DBTor, multi-thread support could be used for speculative translation and other system management tasks, e.g., Tcache handling, peripherals and interrupt polling, among others.

- **Real-time requirements** would be extremely hard to obtain due to the DBT overheads and source/target clock domains' disparity. This requirement is rarely addressed in cross-ISA DBT, because the translation overheads are very high, which difficult execution timing monitoring. Moreover, when using an IR-based DBTor, an one-to-multiple instruction translation is commonly used, which obfuscates the timing ratios of the source-to-target instructions matching.

- **Manual decoding and porting of the IR** for the source and target ISAs, respectively, despite functional, revealed to be a fastidious work and error-prone task. Although these options might have saved some time during the early development of the DBTor, they are fully NRE and not reusable, and in case of a new source or target ISA porting, they would have to be repeated.

- **Self-modifying code support** features were not addressed. Although the mechanisms to deal with it were available, since the source architecture does not allow its practice, it was not a concern during the deployment.
- **BB chaining** optimization is also a simple technique to implement, but due to time constraints, it was not incorporated in the system.
- **Cold code emulation** was not included in the translator. This feature takes advantage of the decoding effort of the instructions to quickly emulate their behavior during their first iteration. After a BB translation, the DBT engine does not require to switch to its execution, since it was emulated already. However, if the same BB is called again for execution, its cached translation will be natively executed.
- **Direct register mapping** was also avoided for DBT portability reasons. This option harms the performance because replacing quick native register operations by load-store operations, will incur in a greater penalty.
- **Timer peripherals** pose some challenges on its support. The configuration sniffing and remapping of the source timers to target timers is supported, as long as the code to map the equivalent configurations is added to the translator. However, and due to differences of clock domains, performance penalties and general de-synchronization between the native and translated execution environments, this topic requires further research in order to be supported.

8.3 Future Work

Despite the contributions this thesis provides to the state of the art, improvements in terms of limitations and research guides are indicated as future work in this section. The proposed tasks are presented as:

- **Energy consumption profiling** should be addressed in the future, measuring the power consumption data, and proceeding to its analysis, in order to establish efficiency ratings for the different DBT components' variations.
- **Make the DBTor available as a service** for OS integration. This suggestions targets not only the integrations of the DBT engine in a OS, as a binaries compatibility service, but also the re-design of the engine for a

multi-thread environment. This deployment might favor future optimization techniques exploration, e.g., speculative translation efforts, code optimizations, and so on.

- **Self-modifying code support** features, previously addressed as a limitation, should be included and tested in the system. It is suggested a full cache eviction strategy, when a write to the program memory is detected, in order to avoid code coherency issues.

- **BB chaining** optimization, also mentioned in the Section Limitations, should also be implemented in the system. This technique should provide considerable performance gains, while having little impact on the deployment, since a full eviction Tcache policy poses no coherency issues to BB chaining.

- **Cold code emulation** is another recommended feature to the DBTor. Despite code translation being generally preferred over code emulation, the use of this technique during the decode stage should result in pure performance gains. This results from the capitalizing of great part of the decoding effort, while avoiding Translation to Execution context switch and the native execution of the BB's first iteration.

- **Direct register mapping** technique should also be implemented, while maintaining portability flexibility. It is suggested the adoption of an adaptable strategy that directly maps the source registers to the available target machine registers and allocates the remaining registers to memory locations.

- **Automated ISA pairing tool** for easy source and target portings of the DBTor, is a desirable feature to be included in the DBT framework. This tool should address the NRE efforts minimization in three ways:

  1. Source ISA to IR mapping. This feature's goal is to achieve an automated mapping of the source ISA to the IR micro operations. This might be achieved through a meta-representation of the ISAs instruction, for functionality, operands and operations identification and correspondence with the IR's micro operation. Although this feature would not dismiss human intervention in the ISA's meta-representation description, it would greatly reduce the necessary knowledge of the ISA for new portings.

  2. IR to target ISA porting. Similarly to the previous feature, this one should achieve an automated porting of the IR micro operations to

target machine code. This could be attained either by employing compiler techniques, like in QEMU, or through more advanced techniques, like the ones resourcing the LLVM compiler infrastructure.

3. Automatic decoding generation. The goal of its implementation would be to attain an automatic correspondence of the source ISA instructions to the target machine code, combining the functionalities of 1. and 2., into a decoding algorithm. The resulting algorithm, expressed as a C++ method for post class integration, should explore advanced, yet efficient, decoding methods. For instance, binary trees can be used to obtain efficient code generation. The automatic decoding tool should also explore possible direct source to target instruction correspondence, in order to avoid IR overheads.

- **Additional sources and target ISA support** should be added to the FAT-DBT to test its portability and for enhanced application possibilities. MIPS architecture poses a good target candidate because of its regular ISA, while any of the legacy ISAs mentioned in Chapter 2 are possible source candidates to be supported.

- **Additional hardware support** to assist and accelerated the DBTor should also be explored. On a short term, it is suggested the full CC evaluation in hardware and a decoding algorithm porting to hardware. Nonetheless, many other tasks are accountable for hardware offloading when FPGA fabric is available in the target device.

- **FAT-DBT enhancement** by improving the GUI to ease system configuration and reduce the number of code generation steps. The DSL semantics should also being improved, with the usage of semantic technology to describe the domain knowledge. A semantically enhanced DSL will improve the model validation and reduce the elaboration development efforts. This will contribute to greater system scalability, configuration granularity, code generation efficiency and design verification.

8.4 List of Publications

The following publications resulted from the work developed in this thesis:

"A Dynamic Binary Translator Architecture for Resource-Constrained Embedded Systems", F. Salgado, T. Gomes, J. Cabral and A. Tavares, [UNDER REVIEW] in IEEE Transactions on Computers.

"MODELA DBT: Model-Driven Elaboration Language Applied to Dynamic Binary Translation", F. Salgado, A. Martins, D. Almeida, T. Gomes, J. L. Monteiro, and A. Tavares, in IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society, Beijing, 2017.

"Condition Codes Evaluation on Dynamic Binary Translation for Embedded Platforms", F. Salgado, T. Gomes, S. Pinto, J. Cabral and A. Tavares, in IEEE Embedded Systems Letters, vol. 9, no. 3, pp. 89-92, Sept. 2017.

"Leveraging On-Chip Debug Features for Condition Codes Handling in Dynamic Binary Translation", F. Salgado, J. Mendes, A. Tavares and M. Ekpanyapong, in 2012 Brazilian Symposium on Computing System Engineering, Natal, 2012.

"Shifting SOA to MPSoC: An exploratory example of application", F. Salgado et al., in Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012), Krakow, 2012, pp. 1-4.

Bibliography

[1] Microsemi Corporation, "SmartFusion2 SoC FPGA Advanced Development Kit." [Online]. Available: https://www.microsemi.com/products/fpga-soc/design-resources/dev-kits/smartfusion2/smartfusion2-advanced-development-kit, Date accessed September, 10 2017.

[2] Patterson, David, "The Future of Computer Architecture." A white paper prepared for the Computing Community Consortium committee of the Computing Research Association, 2008.

[3] Intel Corporation, "Enabling consistent platform-level services for tightly coupled accelerators." Online, 2008. [Online]. Available: https://www.intel.sg/content/dam/doc/white-paper/quickassist-technology-aal-white-paper.pdf, Date accessed September, 1 2017.

[4] R. Obermaisser, H. Kopetz, and C. Paukovits, "A cross-domain multiprocessor system-on-a-chip for embedded real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 548–567, 2010.

[5] R. A. Klein and R. Moona, "Migrating software to hardware on FPGAs," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pp. 217–224, IEEE, 2004.

[6] M. Oyamada, F. R. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya, "Software performance estimation in MPSoC design," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pp. 38–43, IEEE Computer Society, 2007.

[7] S. Sharma, C. Mukherjee, and A. Gambhir, "A Comparison of Network-on-chip and Buses," in *Proceedings of National Conference on Recent Advances in Electronics and Communication Engineering (RACE-2014)* (P. Kidwelly, ed.), pp. 28–29, 2014.

[8] P. Pampagnin, P. Moreau, R. Maurice, and D. Guihal, "Model driven hardware design: One step forward to cope with the aerospace industry needs," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pp. 179–184, IEEE, 2008.

[9] M. Meier, M. Engel, M. Steinkamp, and O. Spinczyk, "LavA: An open platform for rapid prototyping of mpsocs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 452–457, IEEE, 2010.

[10] G. Savaton, J. Delatour, and K. Courtel, "Roll your own hardware description language," in *OOPSLA & GPCE Workshop Best Practices for Model Driven Software Development*, 2004.

[11] X. Wang, S. X. Hu, E. Haq, and H. Garton, "Integrating legacy systems within the service-oriented architecture," in *Power Engineering Society General Meeting, 2007. IEEE*, pp. 1–7, IEEE, 2007.

[12] "STM32 32-bit ARM Cortex MCUs." STM product web site, 2017. [Online]. Available: `http://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html`.

[13] X. Chen, Z. Zheng, L. Shen, W. Chen, and Z. Wang, "GSM: An Efficient Code Generation Algorithm for Dynamic Binary Translator," *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, pp. 231–235, Dec. 2011.

[14] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis, "The University of Queensland Binary Translator (UQDBT) Framework," *The University of Queensland, Sun Microsystems, Inc*, 2001.

[15] V. Moya, "Study of the techniques for emulation programming," Master's thesis, Universidad Politécnica de Cataluña. España, 2001.

[16] R. Hookway and M. Herdeg, "Digital FX! 32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, pp. 3–12, 1997.

[17] H. Guan, B. Liu, T. Li, and A. Liang, "Multithreaded Optimizing Technique for Dynamic Binary Translator CrossBit," *2008 International Conference on Computer Science and Software Engineering*, pp. 945–952, 2008.

[18] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 191, IEEE Computer Society, 2003.

[19] J. Cui, J. Pang, Z. Shan, and X. Liu, "The multi-threaded optimization of dynamic binary translation," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, vol. 4, pp. 2432–2436, IEEE, 2011.

[20] K. K. Daniel J. Magenheimer, Arndt B. Bergh and J. A. Miller, "Hp 3000 emulation on hp precision architecture computers," *HP Journal*, pp. 87–89, 12 1987.

[21] C. May, "Mimic: a fast system/370 simulator," *SIGPLAN '87 Papers of the Symposium on Interpreters and interpretive techniques*, vol. 22, no. 7, 1987.

[22] K. Andrews and D. Sand, "Migrating a CISC computer family onto RISC via object code translation," *ASPLOS V Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, vol. 27, no. 9, pp. 213–222, 1992.

[23] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communications of the ACM*, vol. 36, no. 2, pp. 69–81, 1993.

[24] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary translation," *Communications of the ACM*, vol. 36, pp. 69–81, feb 1993.

[25] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Fast Simulation of Computer Architectures*, pp. 5–46, Springer, 1995.

[26] Brethes, Mathieu, "Atome-Binary Translation for Accurate Simulation," September 2004. [Online] Available: `http://www8.cs.umu.se/education/examina/Rapporter/MathieuBrethes.pdf`, Date accessed September 15th 2017.

[27] J.-Y. Chen, W. Yang, T.-H. Hung, C. Su, and W. C. Hsu, "On static binary translation and optimization for arm based applications," in *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*, vol. 4, pp. 3–1, 2008.

[28] M. Chapman, D. Magenheimer, and P. Ranganathan, "MagiXen: Combining binary translation and virtualization," *Hewlett-Packard Development Company, L.P. Approved*, 2007.

[29] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, p. 2, 2006.

[30] W. Chen, H. Lu, L. Shen, Z. Wang, and N. Xiao, "DBTIM: An Advanced Hardware Assisted Full Virtualization Architecture," *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 399–404, Dec. 2008.

[31] W. Chen, Z. Wang, H. Lu, L. Shen, N. Xiao, and Z. Zheng, "A Hardware Approach for Reducing Interpretation Overhead," *2009 Ninth IEEE International Conference on Computer and Information Technology*, pp. 98–103, 2009.

[32] F. Xu, L. Shen, and Z. Wang, "A Dynamic Binary Translation Framework Based on Page Fault Mechanism in Linux Kernel," in *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 2284–2289, IEEE, jun 2010.

[33] W. Chen, D. Chen, and Z. Wang, "An approach to minimizing the interpretation overhead in Dynamic Binary Translation," *The Journal of Supercomputing*, vol. 61, pp. 804–825, June 2011.

[34] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere, "Evaluation of dynamic binary translation techniques for full system virtualisation on armv7-a," *Journal of Systems Architecture*, vol. 65, pp. 30–45, 2016.

[35] A. D'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján, "HyperMAMBO-X64," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE '17*, pp. 228–241, ACM Press, 2017.

[36] W. Chen, H. Lu, L. Shen, Z. Wang, N. Xiao, and D. Chen, "A Novel Hardware Assisted Full Virtualization Technique," *2008 The 9th International Conference for Young Computer Scientists*, pp. 1292–1297, Nov. 2008.

[37] K. Ebcioğlu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility," *ACM SIGARCH Computer Architecture News*, 1997.

[38] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 529–548, 2001.

[39] Keppel, David, "Transmeta Crusoe Hardware, Software and Development." AMAS-BT - Workshop on Architectural and Microarchitectural Support for Binary Translation, 2009. [Online]. Available: `http://www.xsim.com/papers/crusoe-amasbt2009.pdf`.

[40] Klaiber, Alexander, "The technology behind crusoe processors." Transmeta Technical Brief, 2000.

[41] Keppel, David, "Crusoe Processor Software Optimization Guide," tech. rep., Transmeta Corporation, 8 2001. Software Developer Bulletin.

[42] E. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritts, P. Ledak, D. Appenzeller, and Z. Filan, "Boa: The architecture of a binary translation processor," *IBM Research Report RC 21665*, 2000.

[43] S. Sathaye, P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, *et al.*, "BOA: Targeting multi-gigahertz with binary translation," in *Proc. of the 1999 Workshop on Binary Translation*, pp. 2–11, sn, 1999.

[44] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *Computer*, vol. 33, pp. 54–59, mar 2000.

[45] G. Ottoni, T. Hartin, C. Weaver, J. Brandt, B. Kuttanna, and H. Wang, "Harmonia," in *Proceedings of the 8th ACM International Conference on Computing Frontiers - CF '11*, p. 1, ACM Press, 2011.

[46] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," *PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 46, no. 6, pp. 74–85, 2011.

[47] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," *Proceedings of the IEEE*, pp. 1–10, 2001.

[48] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa, "Retargetable and reconfigurable software dynamic translation," *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pp. 36–47, 2003.

[49] D. L. Bruening, *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 2004. AAI0807735.

[50] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*, pp. 68–78, 2015.

[51] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale, "Hdtrans: an open source, low-level dynamic instrumentation system," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 175–185, ACM, 2006.

[52] S. Sridhar, J. S. Shapiro, and P. P. Bungale, "Hdtrans: a low-overhead dynamic translator," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 135–140, 2007.

[53] M. Payer and T. Gross, "Fast binary translation: Translation efficiency and runtime efficiency," in *2st Workshop on Architectural and Microarchitectural Support for Binary Translation*, sn, 2009.

[54] M. Payer and T. R. Gross, "Generating low-overhead dynamic binary translators," *Proceedings of the 3rd Annual Haifa Experimental Systems Conference on - SYSTOR '10*, p. 1, 2010.

[55] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers, "Evaluating indirect branch handling mechanisms in software dynamic translation systems," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pp. 61–73, IEEE Computer Society, 2007.

[56] N. Kumar and B. Childers, "Flexible instrumentation for software dynamic translation," in *Workshop on Exploring the Trace Space*, 2003.

[57] N. Kumar, C. Bruce R., D. Williams, J. W. Davidson, and M. L. Soffa, "Compile-time planning for overhead reduction in software dynamic translators," *International Journal of Parallel Programming*, vol. 33, pp. 103–114, Jun 2005.

[58] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa, "Overhead reduction techniques for software dynamic translation," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 200–, April 2004.

[59] J. a. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Reducing pressure in bounded DBT code caches," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems - CASES '08*, p. 109, ACM Press, 2008.

[60] R. W. Moore, J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Addressing the challenges of dbt for the arm architecture," *ACM Sigplan Notices*, vol. 44, no. 7, pp. 147–156, 2009.

[61] J. a. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Enabling dynamic binary translation in embedded systems with scratchpad memory," *ACM Transactions on Embedded Computing Systems*, vol. 11, pp. 1–33, dec 2012.

[62] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, pp. 190–200, June 2005.

[63] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 271–282, IEEE Computer Society, 2005.

[64] K. Zhang, T. Zhang, and S. Pande, "Binary translation to improve energy efficiency through post-pass register re-allocation," *Proceedings of the fourth ACM international conference on Embedded software - EMSOFT '04*, p. 74, 2004.

[65] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692–707, 2010.

[66] D. Hong, C. Hsu, C. Chang, and J. Wu, "A Dynamic Binary Translation System in a Client/Server Environment," *iis.sinica.edu.tw*, 2012.

[67] L. Ling, C. Chao, S. Tingtao, L. Alei, and G. Haibing, "DistriBit: A Distributed Dynamic Binary Execution Engine," in *2009 Third Asia International Conference on Modelling & Simulation*, pp. 602–607, IEEE, 2009.

[68] H. Guan, Y. Yang, K. Chen, Y. Ge, L. Liu, and Y. Chen, "DistriBit," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, p. 684, ACM Press, 2010.

[69] W. Chen, L. Shen, H. Lu, Z. Wang, and N. Xiao, "A light-weight code cache design for dynamic binary translation," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pp. 120–125, 2009.

[70] W. Zhang, B. Calder, and D. M. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 87–98, IEEE, 2005.

[71] D. Ung and C. Cifuentes, "Optimising hot paths in a dynamic binary translator," *ACM SIGARCH Computer Architecture News*, vol. 29, pp. 55–65, Mar. 2001.

[72] H. Guan, E. Zhu, K. Chen, R. Ma, Y. He, H. Deng, and H. Yang, "A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation," *Journal of Software*, vol. 6, pp. 2341–2349, Dec. 2011.

[73] H. Guan, E. Zhu, H. Wang, R. Ma, Y. Yang, and B. Wang, "SINOF: A dynamic-static combined framework for dynamic binary translation," *Journal of Systems Architecture*, vol. 58, pp. 305–317, Sept. 2012.

[74] H. Guan, B. Liu, Z. Qi, Y. Yang, H. Yang, and A. Liang, "CoDBT: A multi-source dynamic binary translator using hardware-software collaborative techniques," *Journal of Systems Architecture*, vol. 56, pp. 500–508, oct 2010.

[75] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, and G. Li, "Godson-3: A scalable multicore RISC processor with x86 emulation," *IEEE micro*, pp. 17–29, 2009.

[76] F. Vahid, G. Stitt, and R. Lysecky, "Warp Processing: Dynamic Translation of Binaries to FPGA Circuits," *Computer*, vol. 41, pp. 40–46, jul 2008.

[77] R. a. Sokolov and a. V. Ermolovich, "Background optimization in full system binary translation," *Programming and Computer Software*, vol. 38, pp. 119–126, jun 2012.

[78] H. Guan, R. Ma, H. Yang, Y. Yang, L. Liu, and Y. Chen, "MTCrossBit: A dynamic binary translation system based on multithreaded optimization," *Science China Information Sciences*, vol. 54, pp. 2064–2078, Sept. 2011.

[79] X. Tu, H. Jin, Z. Yu, J. Chen, and Y. Hu, "MT-BTRIMER: A Master-Slave Multi-threaded Dynamic Binary Translator," *2010 Fifth International Conference on Frontier of Computer Science and Technology*, pp. 51–56, Aug. 2010.

[80] C. Wang, V. Ying, and Y. Wu, "Supporting Legacy Binary Code in a Software Transaction Compiler with Dynamic Binary Translation and Optimization," in *Compiler Construction*, vol. 4959 LNCS, pp. 291–306, Springer Berlin Heidelberg, 2008.

[81] Y.-S. Hwang, T.-Y. Lin, and R.-G. Chang, "DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator," *ACM Transactions on Architecture and Code Optimization*, vol. 7, pp. 1–36, Dec. 2010.

[82] Probst, Mark, "Fast machine-adaptable dynamic binary translation," *Proceedings of the Workshop on Binary Translation*, pp. 1–7, 2001.

[83] M. Probst, a. Krall, and B. Scholz, "Register liveness analysis for optimizing dynamic binary translation," *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pp. 35–44, 2002.

[84] Probst, Mark, "Dynamic Binary Translator," in *UKUUG Linux Developers' Conference Linux 2002, 4-7 July 2002, Bristol*, pp. 1–12, 2002.

[85] D. Ung and C. Cifuentes, "Machine-adaptable dynamic binary translation," *ACM SIGPLAN Notices*, vol. 35, pp. 41–51, July 2000.

[86] Bellard, Fabrice, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 41–41, USENIX Association, 2005.

[87] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu, "Hermes: A fast cross-ISA binary translator with post-optimization," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 246–256, IEEE, feb 2015.

[88] Y. Yindong, G. Haibing, and Z. Erzhou, "Crossbit: a multi-sources and multi-targets DBT," *CLOUD COMPUTING 2010 : The First International Conference on Cloud Computing, GRIDs, and Virtualization CrossBit:*, pp. 41–47, 2010.

[89] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, pp. 91–98, ACM, 1994.

[90] Blank, Tom, "A survey of hardware accelerators used in computer-aided design," *IEEE Design Test of Computers*, vol. 1, pp. 21–39, Aug 1984.

[91] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

[92] T. Gomes, P. Garcia, S. Pinto, F. Salgado, J. Cabral, J. Monteiro, and A. Tavares, "Hardware-software extensions to a softcore processor for fpga-based adaptive pid control," in *Industrial Electronics (ISIE), 2013 IEEE International Symposium on*, pp. 1–4, IEEE, 2013.

[93] P. Kinsman and N. Nicolici, "Dynamic binary translation to a reconfigurable target for on-the-fly acceleration," *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 286–287, 2011.

[94] W. Chen, Z. Wang, and D. Chen, "An emulator for executing IA-32 applications on ARM-based systems," *Journal of Computers*, vol. 5, no. 7, pp. 1133–1141, 2010.

[95] G. Kondoh and H. Komatsu, "Dynamic binary translation specialized for embedded systems," *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '10*, p. 157, 2010.

[96] M. Payer, B. Bluntschli, and T. R. Gross, "LLDSAL," in *Proceedings of the seventh workshop on Domain-Specific Aspect Languages - DSAL '12*, p. 15, ACM Press, 2012.

[97] S. Makarov, A. D. Brown, and A. Goel, "An Event-Based Language for Dynamic Binary Translation Frame Works," *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 499–500, 2014.

[98] J. Paredes, *Dynamic binary translation for embedded systems with scratchpad memory.* PhD thesis, University of Pittsburgh, 2011.

[99] Y. Yao, Z. Lu, Q. Shi, and W. Chen, "FPGA based hardware-software co-designed dynamic binary translation system," *2013 23rd International Conference on Field Programmable Logic and Applications, FPL 2013 - Proceedings. IEEE Computer Society.*, pp. 0–3, 2013.

[100] D. Richie and J. Ross, "Cycle-accurate 8080 emulation using an ARM11 processor with dynamic binary translation," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 186–189, IEEE, oct 2014.

[101] Intel Corp., "MCS-51 Microcontroller Family User's Manual," 1994. [Online] Available: http://www.keil.com/dd/docs/datashts/intel/ism51.pdf Date accessed September, 11 2017.

[102] Microchip, "8-bit PIC Microcontrollers," 2002. Product page: http://www.microchip.com/design-centers/8-bit, Date accessed September, 11 2017.

[103] Motorola, "M68HC11 Reference Manual." [Online]. Available: https://www.nxp.com/docs/en/reference-manual/M68HC11RM.pdf, Date accessed September, 11 2017.

[104] Zilog, "Z80 Family Cpu User Manual," 2002. User Manual UM008003-1202.

[105] ARM Limited, "ARM Architecture Thumb-2 Supplement," 2005. Reference Manual.

[106] ARM Limited, "ARM CoreSight Architecture Specification," 2013.

[107] Atmel, "AVR Microcontrollers AVR Instruction Set Manual Table of Contents," *AVR Microcontrollers*, 2016.

[108] Imagination Technologies, "MIPS Architecture For Programmers Volume I-B : Introduction to the microMIPS32 Architecture," tech. rep., Imagination Technologies, 2011.

[109] EECatalog, "Engineers Guide to 8-Bit, 16-Bit & 32-Bit Technologies," *Embedded Systems Engineering*, 2016.

[110] DOLPHIN Integration, "The future of the 8051 legacy upgraded for the Internet of Things (IoT)," *Embedded Systems Engeneering*, pp. 1–11, 2015.

[111] A. Limited, "ARM v7-M Architecture Reference Manual," tech. rep., ARM Limited, 2010.

[112] ARM Limited, "Procedure call standard for the arm architecture," Tech. Rep. ARM IHI 0042F, ARM Limited, 2015.

[113] Microsemi Corporation, "SmartFusion2 product page." Microsemi website, 2017. [Online]. Available: https://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion2, Date accessed September 11th 2017.

[114] Microsemi Corporation, "SmartFusion2 Microcontroller Subsystem User Guide." User Guide, 2013.

[115] Microsemi Corporation, "Libero SoC v11.4 User's Guide." User Guide, 2012.

[116] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms." Online: `http://arxiv.org/abs/1308.5174`, 2013.

[117] R. Guide and M. Architecture, "IAR C / C ++ Compiler Reference Guide for the 8051 Microcontroller Architecture," tech. rep., IAR Systems, 2011.

[118] H. Guan, H. Yang, Z. Qi, Y. Yang, and B. Liu, "The Optimizations in Dynamic Binary Translation," *2010 Proceedings of the 5th International Conference on Ubiquitous Information Technologies and Applications*, pp. 1–6, Dec. 2010.

[119] N. Cardoso, *Middleware e ferramentas para desenvolvimento de sistemas de vigilância para segurança, controlo e conforto (SVSC 2-M Toolkit).* PhD thesis, Universidade do Minho Escola, 2013.

[120] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," *SIGPLAN Not.*, vol. 31, pp. 324–341, Oct. 1996.

[121] M. F. Fernandez, "Simple and effective link-time optimization of modula-3 programs," in *PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, vol. 30, ACM, 1995.

[122] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," *ACM Sigplan Notices*, vol. 31, no. 10, pp. 324–341, 1996.

[123] Troy D. Hanson, Arthur O'Dwyer, "Uthash, A hash table for C structures." [Online]. Available: `https://troydhanson.github.io/uthash/`, Date accessed July, 30 2017.

[124] C. Chao, Z. Yuyu, G. Haibing, and L. Alei, "A Two-Phase Optimization Approach for Condition Codes in a Machine Adaptable Dynamic Binary Translator," *2009 WRI World Congress on Computer Science and Information Engineering*, pp. 29–32, 2009.

[125] Philips Semiconductors, "80C51 family programmer s guide and instruction set," tech. rep., Philips Semiconductors, 1997.

[126] Microchip Technology, "PICmicro Mid-Range MCU Family Reference Manual," Tech. Rep. December 1997, Microchip, Inc., 1997.

[127] E. Borin and Y. Wu, "Characterization of dbt overhead," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 178–187, IEEE, 2009.

[128] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-Aware Interrupt Controller: Priority Space Unification in Real-Time Systems," *IEEE Embedded Systems Letters*, vol. 7, pp. 27–30, mar 2015.

[129] T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares, "Bringing Hardware Multithreading to the Real-Time Domain," *IEEE Embedded Systems Letters*, vol. 8, pp. 2–5, mar 2016.

[130] R. Domínguez, D. Schaa, and D. Kaeli, "Caracal : Dynamic Translation of Runtime Environments for GPUs." AMAS-BT 2011 - AMAS-BT: 4th Workshop on Architectural and Microarchitectural Support for Binary Translation, 2011.

[131] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 1208–1213, 2008.

[132] H.-S. Kim and J. Smith, "Hardware support for control transfers in code caches," in *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*, pp. 253–264, IEEE Comput. Soc, 2003.

[133] K. Hazelwood and R. Cohn, "A Cross-Architectural Interface for Code Cache Manipulation," in *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 17–27, IEEE, 2006.

[134] F. Salgado, "M$^2\mu$P (multithreading microprocessor) memory hierarchies implementation," Master's thesis, Universidade do Minho/Asian Institute of Technology, Portugal/Thailand, 5 2011.

[135] ARM Limited, "Amba 3 ahb-lite protocol specification," Tech. Rep. ARM IHI 0033A, ARM Limited, 2006. http://www.arm.com.

[136] N. Cardoso, P. Rodrigues, J. Vale, P. Garcia, P. Cardoso, J. Monteiro, J. Cabral, J. Mendes, M. Ekpanyapong, and A. Tavares, "A generative-oriented model-driven design environment for customizable video surveillance systems," *EURASIP Journal on Embedded Systems*, vol. 2012, p. 7, dec 2012.

[137] ARM and Qualcomm Technologies, Inc., "Enabling the Next Mobile Computing Revolution with Highly Integrated ARMv8-A based SoCs," tech. rep., Qualcomm Technologies, Inc., 2014.

[138] Bettini, Lorenzo, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[139] Voelter, Markus, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2010-2013.

[140] Fowler, Martin, *Domain Specific Languages*. Addison-Wesley Professional, 1st ed., 2010.

[141] Czarnecki, Krzysztof, "Overview of Generative Software Development," in *Unconventional Programming Paradigms: International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers* (J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, eds.), pp. 326–341, Springer Berlin Heidelberg, 2005.

[142] M. V. Thomas Stahl, *Model-Driven Software Development*. John Wiley & Sons, Ltd, 2006.

[143] J.-P. Tolvanen and M. Rossi, "MetaEdit+: Defining and Using Domain-specific Modeling Languages and Code Generators," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pp. 92–93, ACM, 2003.

[144] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*. Independent Technology Guides, Pearson Education, 2009.

[145] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, and S. Zarnekow, "Xtext User Guide." `http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf`, 2010. Accessed: 2017-03-11.

[146] Eclipse.org, “Xtext - Language Engineering Made Easy!.” [Online] Available: https://www.eclipse.org/Xtext/, Date accessed November, 3 2017.

[147] Eclipse.org, “Xtend Introduction.” [Online] Available: http://www.eclipse.org/xtend/documentation/, Date accessed November, 3 2017.